国外经典教材

PEARSON Prentice Hall

Data structures and Program Design In C++

C++数据结构与程序设计

(美) Robert L. Kruse
Alexander J. Ryba 著
钱丽萍 译

清华大学出版社

Data structures and Program Design In C++

本书系统地介绍了数据结构的内容及程序设计的方法。各章首先介绍核心论题，随后给出示例、应用和案例研究，并用面向对象的程序设计语言C++实现示例。每一章后都附有程序设计规则、启示和易犯的错误，且配有习题和程序设计项目。本书精心设计了一些具有启发性的例子，在对例子进行详细的解释后，采用更通用的形式来表达出各种数据结构所蕴含的思想。此外，各章最后附有供进阶学习本章知识点所用的阅读参考资料，有利于读者加深理解。书后附录包括了算法分析中用到的数学知识、随机数的生成方法、软件包和实用函数，并按主题总结了分布在各章中的程序设计规则、启示和易犯的错误。

ISBN 7-302-07804-1

9 787302 078043 >

定价:59.00元

PEARSON Prentice Hall

文稿编辑：汤涌涛
封面设计：久久度企划
读者信箱：Book@21bj.com
信息网站：http://www.ePress.cn
http://www.34.cn
http://www.pearsoned.com

国外经典教材

C++数据结构与程序设计

（美） Robert L. Kruse
Alexander J. Ryba 著

钱丽萍 译

清华大学出版社
北 京

内 容 简 介

这本精心制作的课本结合面向对象程序设计和 C++强有力的特性，构建数据结构的基本思想，设计了程序和有趣的应用。在此过程中，本书探讨了作为软件设计基本工具的问题求解和设计原理、数据抽象、递归和算法的比较分析。本书使用真实的案例研究、可重用的软件开发和程序设计项目来增强理解。

本书内容详尽且配有大量的实例和习题。书中所有算法都做了详细的注解，有利于读者理解算法的实质和编程思想。

本书既可作为高等学校计算机及相关专业学生的教材，亦可供计算机应用领域的工程技术人员参考，尤其适合于应用 C++语言编程的科技人员。

Simplified Chinese edition copyright © 2003 by **PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.**
Original English language title from Proprietor's edition of the Work.

Original English language title: Data Structures and Program Design In C++, 1st Edition by Robert L. Kruse, Alexander J. Ryba , Copyright © 1999
EISBN: 0-13-768995-0
All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc.. publishing as Prentice-Hall, Inc.
This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).
本书中文简体翻译版由 Pearson Education 授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。

北京市版权局著作权合同登记号 图字：01-2003-0566

本书封面贴有 Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

C++数据结构与程序设计/ （美）克鲁斯（Kruse, R. L.），（美）瑞贝（Ryba, A. J.）著；钱丽萍译. —北京：清华大学出版社，2003
（国外经典教材）
书名原文：Data Structures and Program Design In C++
ISBN 7-302-07804-1
Ⅰ.C… Ⅱ.①克… ②瑞… ③钱… Ⅲ.① C 语言—程序设计—教材 ② 数据结构—教材 Ⅳ. ① TP312 ② TP311.12
中国版本图书馆 CIP 数据核字（2003）第 116041 号

出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969

文稿编辑：汤涌涛
封面设计：久久度企划
印 装 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：37.5　**字数**：908 千字
版　　次：2004 年 1 月第 1 版　2004 年 1 月第 1 次印刷
书　　号：ISBN 7-302-07804-1/TP·5688
印　　数：1～4000
定　　价：59.00 元

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770175-3103 或(010)62795704。

读者对本书的赞誉

★★★★★

本书以前的 Pascal 版本特别易于阅读，也是我学习数据结构的用书。课本中包含的许多图表对于理解算法来说其价值是无法衡量的。本书对以前的版本做了进一步改进，加入了更直观的图片，并对旧书进行了更新，加入了旧书编写时未出现的一些新的算法和分析技术，如平摊分析。作者的编写风格完全易于理解。我收集有多本有关算法和数据结构的书，而这一本是迄今为止可读性最好的。

—— Jeffrey Hsu，摘自亚马逊网站

★★★★★

我个人认为，作为数据结构的入门书籍，这本书是极好的。首先，它通过有趣而具体的例子成功地激发了学生学习相关主题的兴趣。其次，它注意使用图画和色彩，从而使得阅读尽可能容易。最后，作者在程序中精细地“提升”了“结构”的概念并在整篇课文中都说明了它的重要性。

——一名读者，摘自亚马逊网站

★★★★★

本书对那些需要研究构成通用程序的数据结构的 C 程序设计新手来说是优秀的资源。例如课文以一定的深度讨论了队列、栈、哈希表和递归，但这些讨论却是易于理解的。

——一名读者，摘自亚马逊网站

★★★★★

如果你是一个中级C++程序设计者，则本书以一种清晰的方式介绍了数据结构，书中有一些我在教科书中曾看到过的最好的图形。每一章的结尾都包含了非常好的练习和高级阅读物的列表（出色的特征）。也许对于高级学生来说，这本书显得不够详细，但我确实非常喜欢这本书。非常感谢作者！

—— 摘自亚马逊网站

前　言

木匠学徒可能仅仅需要一把斧头和一把锯子，而建筑师却使用许多精密的工具。计算机程序设计同样需要完善的工具来应对实际应用的复杂性，而只有不断使用这些工具进行实践，才能积累使用技能。本书将结构化问题求解、面向对象的程序设计、数据抽象以及算法的比较分析看作程序设计的基本工具。书中详细设计了几个相当规模的案例研究，以此说明如何同时使用所有这些工具来建立完整的程序。

我们所研究的许多算法和数据结构拥有内在的简洁性，这是一种掩盖了它们的应用范围和能力的简洁性。用不了多长时间，学生即发现在预备课程中经常使用的一些天真的方法得到了巨大的改进。然而方法的这种简洁性却被不确定性淡化了。学生很快又发现，几种方法中的哪一种在特定应用中能证明是最佳的，其答案远非显而易见。因此我们利用这个机会，及早地介绍那些既具有内在趣味性，又具有实际重要性的真正复杂的问题，并展示数学方法在算法验证和分析中的应用。

许多学生感到很难将抽象的概念转化为实践。因此，本书特别关注了将概念表达成算法和将算法细化成能运用于实际问题的具体程序。类似地，本书将数据说明和抽象的过程放在数据结构选择及其实现之前。

精心设计启发性的例子，继之以更通用的形式来表达这个概念，我们认为这是一个从具体到抽象的发展过程。大多数学生在其学业的早期，需要通过目睹所学概念的直接应用来加以巩固，他们需要编写并且运行程序，以阐明自己所学的每个重要概念。因此本书包含许多例子程序，有较短的函数，也有相当长的完整程序。另外，习题和程序设计项目构成了本书不可缺少的部分。其中许多习题和项目是所学主题的直接应用，经常要求编写和运行程序，以此可以测试和比较算法。有些是比较大的项目，并且有一些项目适合学生小组共同设计完成。

本书中的程序采用流行的面向对象语言C++编写。我们认为许多面向对象的技术为数据结构设计的基本原理提供了自然的实现。这样，C++允许我们构造安全、有效和简单的数据结构实现。我们承认C++很复杂，学生们需要通过学习一门数据结构课程来深化和提炼他们对这门语言的理解。在编写本书的过程中，我们尽量通过仔细介绍和解释C++各种面向对象的特征来支持这种深化。因此，我们在第 1 章开始时假设读者能自如地使用C++的基本部分(实质上是 C 的子集)，并逐步地加入C++的面向对象的元素，如类、方法、构造函数、继承、动态内存管理、析构函数、拷贝构造函数、重载函数和运算符、模板、虚函数和 STL。当然，本书重点讲述的是数据结构，因此对不太了解C++的学生来说，阅读本书前需要先学习一本C++程序设计的课本。

大纲

* * * 程序设计原理

第 1 章通过完成首个大型项目(Conway 的 Life 游戏)，详细说明面向对象的程序设计、

自顶向下的细化、复查和测试的原理，向学生展示这些原理，并且希望他们从头到尾都遵循这些原理。同时，这个项目为学生提供了一个机会来复习C++基本特征的语法，C++是本书通篇使用的程序设计语言。

* * *栈

第2章介绍我们要学习的第一种数据结构——栈。本章将栈应用到反转输入、模拟桌面计算器和检查括弧嵌套等程序设计中。开始时利用STL栈实现，随后设计和使用我们自己的栈实现。第2章的一个主要目的是使学生理解信息隐藏、封装和数据抽象这些概念后面的思想，并将自顶向下的设计方法应用于数据和算法。本章最后对抽象数据类型进行了介绍。

* * *队列

队列是第3章的中心论点。本章详细说明抽象数据类型的几种不同实现，并设计一个大型的应用程序来说明不同实现的相对优点。本章介绍了继承这一重要的面向对象技术。

* * *链栈和队列

第4章提供栈和队列的链式实现。本章首先对C++中的指针和动态内存管理做一个全面的介绍。在展示一种简单的链栈实现后，我们讨论析构函数、拷贝构造函数和重载赋值运算符，它们都是链式结构的安全的C++实现中所需要的。

* * *递归

第5章通过研究栈与递归问题的求解和程序设计之间的关系来继续阐明栈。本书通过研究几个实质性的递归应用来巩固这些概念，这些应用包括回溯和树结构的程序。如果需要的话，本章可以安排在此处之前、第2章结束后的任何时候学习。

* * *表和字符串

第6章的主题是以链式和邻接实现的更通用的表。本章也包含封装的字符串的实现、C++模板介绍和对算法分析的非形式化的介绍。

* * *查找

* * *排序

* * *表和信息检索

第7章、第8章和第9章分别介绍查找、排序和表格访问(包含散列)的算法。这些章节阐述算法和相关抽象数据类型、数据结构及实现之间的相互关系。课本中为初步的算法分析引入"大-O"和相关符号，并强调在空间、时间和程序设计强度的最佳利用方面作出关键的选择。这些选择要求我们找到分析方法来评价算法，而进行这些分析犹如一场战斗，需要利用组合数学这样的武器。在初级水平，我们既不期望学生全面掌握，也不期望他们拥有完备的数学知识而使自己的技能变得完美。因此，我们的目的是帮助学生认识到这些技能的重要性，期待以后有机会再学习数学。

* * *二叉树

二叉树肯定是数据结构中最精致和有用的数据结构之一。第10章将对它进行研究，并将它与表、查找、排序中的概念相结合。作为递归定义的数据结构，二叉树为学生灵活地在数据结构和算法中应用递归提供了条件。本章以基本主题开始，并进展到诸如伸展树和平摊算法分析这样的高深的主题。

＊＊＊多路树

第11章继续学习更复杂的数据结构，包括Trie树、B-树和红-黑树。

＊＊＊图

第12章介绍图，将它作为对问题求解有用的更通用的结构，并介绍图中最短路径和最小生成树的一些经典算法。

＊＊＊案例研究：波兰表示法

第13章中的案例研究相当详细地分析了波兰表示法，以此研究递归、树和栈作为问题求解和算法设计工具的相互关系。所涉及的一些问题能充当编译器设计的一个非形式化的介绍。我们照例将这个算法完全设计成一个能运行的C++程序，此程序接受输入普通(中缀)形式的表达式，将它翻译成后缀形式，并用指定的变量值评估此表达式。第13章可安排在10.1节完成后的任何时候学习。

附录中的几个论题并非切合于本书主题，但学生在课程准备过程中经常遗漏它们。

＊＊＊数学方法

附录A介绍了离散数学中的一些论题。它的最后两节是斐波纳契数(Fibonacci number)和卡塔兰数(Catalan number)，这两节更为复杂，并且并非课本中任何重要论题所必需的，但附录中将它们包含进来是鼓励更偏向数学的综合兴趣。

＊＊＊随机数

附录B讨论伪随机数、生成器和应用，这是许多学生感兴趣的论题，但它不适合放在本课程的正文中。

＊＊＊软件包和实用函数

附录C对本书中设计和多次使用的各种实用程序和数据结构软件包进行分类。附录C中讨论了声明和定义文件、变换单位、本书中通篇所用的实用程序包和一个计算CPU时间的软件包。

＊＊＊程序设计技术规则、启示和易犯的错误

最后，附录D收集了所有的程序设计技术规则及分散在本书中的所有启示和易犯的错误，并将它们按主题组织以方便参考。

课程结构

＊＊＊必备条件

学习本书的必备条件是学习过一门基本的程序设计课程，具有使用C++基本特征的经验。然而，因为我们仅仅是逐步谨慎地引入复杂的C++技术，我们相信，结合使用一本补充的C++课本和关于C++语言问题的更多的用法说明和重点，加之本书提供了C++版的数据结构教程，即使对于程序设计背景是另一门语言(如C, Pascal或Java)的学生，这本书仍然适用。

中学数学的良好知识能满足几乎所有的算法分析的需要，但进一步的(可能是同步的)离散数学的知识准备将颇有价值。附录A中复习了所有要求的数学知识。

＊＊＊内容

本书供诸如ACM Course CS2(程序设计和实现)、ACM Course CS7(数据结构和算法

分析)或结合了这些课程的某门课程使用。本书完全覆盖了 ACM/IEEE 关于数据结构和算法的大多数知识单元[①],它们包括:

AL1 基本数据结构,如数组、表、栈、队列、树和图;

AL2 抽象数据类型;

AL3 递归和递归算法;

AL4 使用大 Oh 符号的复杂性分析;

AL6 排序和查找;

AL8 带有大型案例研究的实际问题求解策略。

本书不论述 3 个最高级的知识单元:AL5(复杂性类,NP-完全问题)、AL7(可计算性和不可判定性)和 AL9(并行和分布式算法)。

本书中大多数章节都是首先介绍核心主题,然后给出例子、应用和更大的案例研究。因此,如果时间仅允许简要学习某个主题,则可以在不失连续性的情况下,仅学习核心主题,快速地从一章跳到另一章。然而当时间允许时,学习那些附加的主题和完成的项目都将使学生和教师从中获益。

* * * 两学期课程

基本上可以用两学期的时间学完本书,学完本书后您就可以了解问题求解、数据结构、程序开发和算法分析等领域的许多论题。学生需要时间和实践来理解通用方法。通过将数据抽象、数据结构和算法的学习同它们在现实规模的项目中的实现相结合,一门综合的课程可以奠定一个坚实的基础,此后可以基于它学习更多的理论课程。即使本书没有完全覆盖它们,但也提供了足够的深度,使得感兴趣的学生在以后的工作中可以继续用它作为参考书目。无论如何,分配主要的程序设计项目并给予足够的时间来完成它们,这一点是重要的。

补充材料

除本书全部内容以外,还有一些补充材料,包括:

- 来自课本的所有的软件包、程序和其他C++代码段,其格式便于按需并入其他程序;
- 课本中一些示例程序和几乎所有程序设计项目的可执行的版本(DOS 或 Windows 平台);
- 课本中每节的提要或总结,适于用作学习指南。

这些材料可通过 ftp 方式获取:以 anonymous 用户登录进站点 *ftp. prenhall. com*,并转到目录 *pub/esm/computer_science. s-041/kruse/cpp*。

致谢

这些年来,本书的 Pascal 版和 C 语言版得到许多人的帮助,他们是家人、朋友、同事和学生,在以前的书中曾特别提到过其中一些人。其他的许多人在学习这些书或是其各种语言

① 参见 Computing Curricula 1991: *Report of the ACM/IEEE-CS Joint Curriculum Task Force*, ACM Press, New York, 1990。

的译本的同时，友好地将他们的意见和建议转发给我们，所有这些都有助于我们将这本书做得更好。

感谢下列审阅者的建议，他们在许多方面帮助我们改进本书。他们是：Vander Linden（加尔文学院）、Jens Gregor（田纳西州大学）、Victor Berry（波士顿大学）、Jeffery Leon（伊利诺伊大学芝加哥分校）、Susan Hutt（密苏里-哥伦比亚大学）、Fred Harris（内华达大学）、Zhi-Li Zhang（明尼苏达大学）和 Andrew Sung（新墨西哥技术学院）。

Alex Ryba 特别感谢 Marquette 大学的 Wim Ruitenburg 和 John Simms，近年来从他们那里获得了富有成效的建议和鼓励，还要感谢以前的学生 Rick Vogel 和 Jun Wang 提出的意见。

Robert Kruse 特别感谢 PreTEX 公司 Paul Mailhot 不断提出忠告并给予帮助，他最初是一名出色的学生，后来作为一名可靠的研究助手，现在已成为一名得力的同事。他在书籍制作的软件开发，项目管理，对出版商、印刷工和作者的问题解答，以及为这项工作的各方面提出忠告和鼓励等方面都做出了重要的贡献。

没有 Prentice Hall 的编辑人员，特别是出版商 Alan Apt、采编 Laura Steele 和主编 Marcia Horton 热心的支持、真诚的鼓励和极大的耐心，这个项目不会开始，当然也不会完成。他们和其他制作人员的帮助是无价的。

Robert L. Kruse

Alexander J. Ryba

目　录

第 1 章　程序设计原理 …… 1
1.1　简介 …… 1
1.2　Life 游戏 …… 3
1.3　程序设计风格 …… 9
1.4　编码、测试和进一步细化 …… 17
1.5　程序维护 …… 29
1.6　结论和复习 …… 34
启示和易犯的错误 …… 39
复习题 …… 39
进阶参考书目 …… 40
第 2 章　栈 …… 43
2.1　栈说明 …… 43
2.2　栈的实现 …… 49
2.3　应用：桌面计算器 …… 57
2.4　应用：括号的匹配 …… 60
2.5　抽象数据类型及其实现 …… 61
启示和易犯的错误 …… 65
复习题 …… 66
进阶参考书目 …… 66
第 3 章　队列 …… 67
3.1　定义 …… 67
3.2　队列的实现 …… 72
3.3　C++队列的循环实现 …… 75
3.4　演示和测试 …… 78
3.5　队列的应用：模拟 …… 81
启示和易犯的错误 …… 93
复习题 …… 93
进阶参考书目 …… 94
第 4 章　链栈和链式队列 …… 95
4.1　指针和链式结构 …… 95
4.2　链栈 …… 107
4.3　带保护的链栈 …… 110
4.4　链式队列 …… 115
4.5　应用：多项式运算 …… 119

4.6 抽象数据类型及其实现 …… 128
启示和易犯的错误 …… 130
复习题 …… 130
第5章 递归 …… 132
5.1 递归导言 …… 132
5.2 递归的原理 …… 142
5.3 回溯法:延缓工作 …… 153
5.4 树结构的程序:在游戏中预测 …… 165
启示和易犯的错误 …… 174
复习题 …… 175
进阶参考书目 …… 176
第6章 表和字符串 …… 177
6.1 表的定义 …… 177
6.2 表的实现 …… 181
6.3 字符串 …… 194
6.4 应用:文本编辑器 …… 201
6.5 数组链表 …… 208
6.6 应用:生成排列 …… 216
启示和易犯的错误 …… 220
复习题 …… 221
进阶参考书目 …… 221
第7章 查找 …… 222
7.1 查找:引言和符号 …… 222
7.2 顺序查找 …… 224
7.3 二分查找 …… 229
7.4 比较树 …… 236
7.5 下限 …… 246
7.6 渐近 …… 250
启示和易犯的错误 …… 260
复习题 …… 260
进阶参考书目 …… 261
第8章 排序 …… 262
8.1 引言和符号 …… 262
8.2 插入排序 …… 264
8.3 选择排序 …… 272
8.4 希尔排序 …… 275
8.5 下限 …… 278
8.6 分而治之排序 …… 281
8.7 链表的归并排序 …… 285

8.8　顺序表的快速排序 …… 292
8.9　堆和堆排序 …… 301
8.10　复习：方法比较 …… 308
启示和易犯的错误 …… 311
复习题 …… 312
进阶参考书目 …… 312
第 9 章　表格和信息检索 …… 314
9.1　引言：突破 lg n 的障碍 …… 314
9.2　矩形表格 …… 315
9.3　各种形态的表格 …… 317
9.4　表格：一种新的抽象数据类型 …… 321
9.5　应用：基数排序 …… 324
9.6　哈希法 …… 329
9.7　关于哈希的分析 …… 340
9.8　结论：方法的比较 …… 345
9.9　应用：再访 Life 游戏 …… 346
启示和易犯的错误 …… 353
复习题 …… 354
进阶参考书目 …… 354
第 10 章　二叉树 …… 356
10.1　二叉树 …… 356
10.2　二叉查找树 …… 368
10.3　建立二叉查找树 …… 384
10.4　高度平衡：AVL 树 …… 392
10.5　伸展树：自我调节的数据结构 …… 407
启示和易犯的错误 …… 428
复习题 …… 429
进阶参考书目 …… 430
第 11 章　多路树 …… 432
11.1　果园、树和二叉树 …… 432
11.2　词典查找树：trie …… 439
11.3　外部查找：B-树 …… 444
11.4　红-黑树 …… 462
启示和易犯的错误 …… 471
复习题 …… 472
进阶参考书目 …… 472
第 12 章　图 …… 474
12.1　数学背景 …… 474
12.2　计算机表示 …… 476

12.3 图的遍历 …… 480
12.4 拓扑排序 …… 482
12.5 贪心算法:最短路径 …… 486
12.6 最小生成树 …… 490
12.7 图作为数据结构 …… 496
启示和易犯的错误 …… 498
复习题 …… 498
进阶参考书目 …… 498
第 13 章 案例研究:波兰表示法 …… 500
13.1 问题 …… 500
13.2 思想 …… 502
13.3 波兰表达式的求值 …… 505
13.4 从中缀式到波兰形式的转换 …… 515
13.5 一个交互式的表达式求值程序 …… 520
进阶参考书目 …… 539
附录 A 数学方法 …… 540
A.1 整数幂的和 …… 540
A.2 对数 …… 542
A.3 排列、组合和阶乘 …… 548
A.4 斐波纳契数 …… 550
A.5 Catalan 数 …… 552
进阶参考书目 …… 555
附录 B 随机数 …… 557
B.1 介绍 …… 557
B.2 策略 …… 557
B.3 程序设计 …… 558
进阶参考书目 …… 562
附录 C 软件包和实用函数 …… 563
C.1 软件包和 C++转换单元 …… 563
C.2 课文中的软件包 …… 564
C.3 实用程序软件包 …… 566
C.4 计时方法 …… 567
附录 D 程序设计规则、启示和易犯的错误 …… 569
D.1 数据结构和算法的选择 …… 569
D.2 递归 …… 572
D.3 数据结构的设计 …… 572
D.4 算法设计和分析 …… 573
D.5 程序设计 …… 574
D.6 用指针对象进行程序设计 …… 575

D.7　调试和测试 …… 576
D.8　维护 …… 576
术语表 …… 578

第 1 章　程序设计原理

本章首先概述良好程序设计的重要原理，特别是它们在大型项目中的应用；然后介绍用于发现有效算法的方法，如面向对象的设计和自顶向下的设计。在此过程中，我们提出将在后继章节中论述的程序设计和数据存储方法方面的问题，并通过使用C++编写程序，复习一下这门语言的一些基本特性。

1.1　简介

编写大型计算机程序的最大困难不在于确定此程序的目标是什么，也不在于找出达到此目标的方法。某个企业的总裁可能会说："让我们用一台计算机来记录所有的存货信息、账目报告和人事文件，并让它告诉我们何时需要再次订购存货，以及何时花费超出了预算线，并且还可以让它处理工资单。"花上充足的时间和精力，系统分析和程序设计员就可以确定企业的各个职员现在是如何完成这些任务的，然后就编写程序用同样的方式来完成这些工作。

＊＊＊大型程序的问题

然而，这种方法几乎必定是灾难性的失败。当系统分析员去调查职员时，他会发现某些工作可以轻易地放到计算机上去执行，于是他就这样做了。然后，当他们将其他工作移到计算机上时，往往发现它依赖于最初的那些任务。遗憾的是，这些任务的输出却或多或少地没有采用正确的格式，因此他们需要更多的程序设计，以将数据从一个任务给定的格式转换成另一个任务需要的格式。这个程序设计项目开始变得像是一床由各色布片拼缀成的被子，其中一些布片较牢，而另一些布片较弱；一些布片被仔细地与邻接的布片缝合，另一些却是勉强带上。如果程序设计员幸运的话，他们的作品可以足够好地连在一起，在大多数时间完成大多数的例行工作。但如果一旦必须做出任何修改，则整个系统将会有不可预测的结果。随后，又会出现新的需求，或者是一个出乎意外的问题，或许甚至是一个紧急事件，而这些程序设计员的成果所具有的功效就好比是用一张拼凑的被单作为从高楼跳下的人的安全网。

本书的主要目的是描述程序设计的方法和工具，这些方法和工具证明对现实规模的项目是有效的，这些程序比用来示范初级程序设计特征的那些普通程序大得多。由于将杂碎的方法用于解决大型问题注定会失败，因此我们首先必须采用一种一致的、统一的和逻辑的方法，也必须仔细遵守程序设计的重要原理。这些原理有时候在编写小程序时被忽略，但对大型项目，忽略它们将证明是灾难性的。

＊＊＊问题说明

处理大型问题时，首个主要障碍就是准确地判断这个问题是什么。有必要将模糊的目标、矛盾的要求和可能未明确说明的需求转换成能够进行编程的、精确规划的项目。人们以前使用的方法或对工作的划分未必最适于机器使用。因此我们的方法必须确定总体目标，

但却是准确的目标，然后渐渐地将工作划分成更小的问题，直到它们达到可管理的规模。

* * * 程序设计

许多程序设计员奉行的准则是："首先让你的程序运行起来，然后使它变得优美。"这一准则对小型程序可能是有效的，但却不适用于大型程序。一个大型程序的每一部分都必须得到妥善地组织、清晰地书写和全面地理解，否则它的结构将被遗忘，也不再能够在以后某个时间连接到项目的其他部分，而那些部分可能由另一个程序设计员来完成。因此我们不把风格从程序设计的其他部分分离出来，但是从一开始我们就必须小心，养成良好的习惯。

* * * 数据结构的选择

即使对非常大的项目，困难通常不在于未能找到一种解决方案，而是在于有太多可用的不同方法和算法，导致我们难以确定其中哪一个最佳、哪一个将导致程序设计困难或者哪一个可能效率低得让人绝望。算法设计中可变性的最大余地通常在于存储程序的数据的办法：

- 它们彼此如何被安排。
- 哪些数据保存在内存中。
- 哪些在需要时被计算。
- 哪些保存在文件中，这些文件又如何安排。

因此，本书的第二个目的是为数据组织和操作提供一些一流的但实质上简单的思想。表、栈和队列是我们最先学习的 3 种结构。随后，我们将为数据处理的重要任务设计一些强大的算法，如排序和查找。

* * * 算法分析

当有若干不同的方法可以用来组织数据和设计算法时，开发一个用来推荐选择的标准就变得很重要了。因此我们将专心分析各种条件下的算法行为。

* * * 测试和验证

调试程序的困难比程序规模的增长要快得多。即，如果一个程序的规模是另一个程序的两倍，则不一定花两倍的时间就能调试它，而很可能会是四倍的时间。许多大型程序（如操作系统）在交付使用时，仍然包含程序设计员无望找出的错误，因为这些困难看起来是难以克服的。有时候已耗时数年努力的项目由于无法发现为何不能工作而废弃。如果我们不希望自己的项目遭受这种命运的话，就必须使用满足以下性质的方法：

* * * 程序正确性

- 减少错误数目，使得更易于发现那些剩余的错误。
- 能够预先验证我们的算法是正确的。
- 能够提供测试程序的方法，使得我们有理由确信这些程序不会行为失常。

这些方法的设计是我们的另一个目的，但它却并非完全在我们的能力可及的范围之内。

* * * 维护

即使在一个程序开发完成、全部调试并交付使用后，仍然需要大量的工作来维护此程序的有效性。在对此程序有新的要求的时候，它的运行环境会改变，必须使之适合新的需求。基于此因，务必使一个大型项目编写得尽可能简单以利于理解和修改。

* * * C++

程序设计语言 C++ 是表达我们将面临的算法的一种十分便利的选择。Bjarne

Stroustrup 在 20 世纪 80 年代早期设计了这门语言，将它作为通俗 C 语言的一种扩展。Stroustrup 在C++中并入的大多数新特性有利于理解和实现数据结构。在C++最重要的特性中，用于我们学习数据结构的有：

* * * 要点

- C++允许数据抽象：这意味着程序设计员能创建新的类型来表达方便其应用程序的任何数据集合。
- C++支持面向对象的设计，在面向对象的设计中，程序设计员定义的类型在算法实现中起到中心作用。
- 重要的是，正如允许面向对象的方法，C++允许使用自顶向下的方法，这也是 C 程序设计员所熟悉的。
- C++便于代码重用及通用目的库的构造。这门语言包含了一个扩展的、有效的和方便的标准库。
- C++改进了 C 语言几个麻烦和危险的方面。
- C++保持了作为 C 语言特点的效率。

灵活性、通用性和高效性的结合，使得C++成为目前程序设计员最流行的选择之一。

我们将发现，C++的数据抽象和面向对象的特性自然地实现了作为数据结构设计基础的普遍原理。因此，我们将仔细解释如何使用C++的这些方面并当它们首次在本书中出现时简要地概述它们的语法（语法规则）。这样，我们将阐明和描述C++中许多不与 C 重叠的特性。关于C++语法的精确细节，请参考C++程序设计方面的课本——我们在本章最后的参考书目中推荐了几本这样的书。

1.2　Life 游戏

冒昧地引用一句古老的格言：实践出真知。

* * * 案例研究

贯穿本章，我们将集中学习一个案例。虽然按现实标准来看它并不大，但它阐明了程序设计的原理和我们应当学习避免的易犯的错误。有时候这个例子会带出通用的原理，有时候我们首先进行综合讨论，但我们的目的始终是发现能在实际应用范围中证明其价值的那些通用原理。在后继章节中，我们将对大型项目使用类似的方法。

我们将使用的例子称为 Life 游戏，由英国数学家 J. H. Conway 在 1970 年提出。

1.2.1　Life 游戏的规则

* * * 定义

Life 游戏实际上是一种模拟，并不是游戏者之间的游戏。它在一个无边界的矩形网格上进行，这个矩形网格中的每个单元可被一个有机体占据，或者不被占据。被占据的单元称为活的，未被占据的单元称为死的。哪一个单元是活的要根据其周围活的邻居单元的数目而一代代地发生变化，规则如下：

* * * 转换规则

(1) 给定单元的邻居是与它在垂直、水平或对角方向上相接的 8 个单元。

(2) 如果一个单元是活的，但没有邻居单元是活的，或者仅有一个邻居单元是活的，则在下一代，此单元就会因孤独而死亡。

(3) 如果一个单元是活的，且有 4 个或 4 个以上的邻居单元也是活的，则在下一代，此单元会因拥塞而死亡。

(4) 一个活的单元，如果具有 2 个或 3 个活的邻居单元，则此单元在下一代仍是活的。

(5) 如果一个单元是死的，则在下一代，如果它不多不少刚好有 3 个邻居单元是活的，则此单元变成活的。所有其他死的单元在下一代仍是死的。

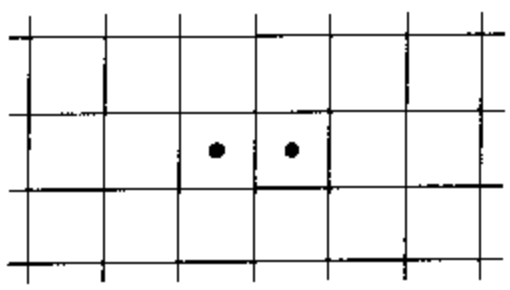
(6) 所有的出生和死亡都刚好在同一时间发生，因此正在死亡的单元有助于另一个单元的出生，但它不能通过减少拥塞而阻止其他单元的死亡；正在出生的单元也不能保护或杀死上一代中活着的单元。

* * * 配置

在一个网格中，活的和死的单元的一种特定的安排称为配置(configuration)。前述的规则解释了一种配置在每一代如何变换成另一种配置。

1.2.2 例子

作为第一个例子，考虑下面这个配置：



每个单元活的邻居的计数如下：

0	0	0	0	0	0
0	1	2	2	1	0
0	1	1	1	1	0
0	1	2	2	1	0
0	0	0	0	0	0

* * * 垂死的例子

由规则(2)，在接下来的一代，两个活的单元都将死亡，而且规则(5)表明没有单元会变成活的，因此这个配置灭绝。

* * * 稳定性

另一方面，下面的配置拥有所示的邻居计数。每个活的单元的邻居单元计数为 3，因此将保持活的，但死的单元都只有 2 个或更少的邻居计数，因此它们当中不会有单元变活。

0	0	0	0	0	0
0	1	2	2	1	0
0	2	● 3	● 3	2	0
0	2	● 3	● 3	2	0
0	1	2	2	1	0
0	0	0	0	0	0

* * * 交替

下面这两个配置将按代交替出现，如配置中给出的邻居计数所指示。

0	0	0	0	0
1	2	3	2	1
1	● 1	● 2	● 1	1
1	2	3	2	1
0	0	0	0	0

和

0	1	1	1	0
0	2	● 1	2	0
0	3	● 2	3	0
0	2	● 1	2	0
0	1	1	1	0

* * * 多样性

这是一个令人惊讶的事实，从十分简单的初始配置，持续许多代后，能发展成 Life 配置的相当复杂进程，而且按代进步时会发生什么变化通常并不明显。一些很小的初始配置会成长为较大的配置，其他的会慢慢灭绝，许多配置会达到一种不变的状态或者每隔几代即经历一种重复模式的状态。

* * * 流行

在 Life 游戏发明后不久，Martin Gardner 在 *Scientific American* 的专栏中对它进行了讨论，从那时起，它已使许多人为之着迷，以至于甚至有一本新闻季刊有好几年都致力于相关的论题。它为家用微型计算机提供了理想的显示。

当然，我们的首要目的是编写一个程序来显示一种初始配置如何逐代变化。

1.2.3　解决方案：类、对象和方法

概括地说，运行 Life 游戏的程序具有这样的形式：

* * * 算法

创建 Life 配置作为活的和死的单元的一种初始安排。

打印 Life 配置。

当用户希望看到下一代时：

　　通过应用 Life 游戏的规则更新配置。

　　打印当前配置。

* * * 类

* * * 对象

在此算法中我们要研究的一件重要的事情是 Life 配置。在C++中，我们使用类(class)来组合数据和用于访问或修改这些数据的方法，数据和方法的这样的组合称为属于给定类的对象。对 Life 游戏来说，我们称这个类为 Life，因此配置 configuration 成为 Life 的一个对象。我们然后对这个对象使用 3 个方法：initialize()将创建具有活的和死的单元的初始配置；print()将打印出当前配置；update()将作出从一代迁移到下一代时发生的所有变化。

* * * C++类

* * * 方法

实际上，每个C++类由表示变量或函数的成员所组成。表示变量的成员称为数据成员，这些成员用于存储数据值。表示某个类的函数的成员称为方法或成员函数，类的方法一般用于访问或改变数据成员。

* * * 客户程序

客户程序即访问某个特定类的用户程序，它能声明并操作那个类的对象，因此在 Life 游戏中，我们声明一个 Life 对象：Life configuration;。

* * * 成员选择运算符

现在可以使用C++运算符.(成员选择运算符)来访问 configuration 的方法。例如，可以如下方式打印出 configuration 中的数据：configuration. print();。

* * * 说明

* * * 信息隐藏

在书写客户程序时，只要我们知道一个C++类的每个方法的说明，即每个方法功能的精确描述，就可以使用这个类，意识到这一点很重要。我们不必知道数据实际上是如何存储的以及方法实际上是如何编程实现的。例如，为了使用 Life 的对象，我们不必确切地知道此对象是如何存储的，或者类 Life 的方法如何完成它们的工作。我们的首个例子反映了一条重要的程序设计策略，此策略称为信息隐藏。

* * * 私有和公用

当开始实现类 Life 时，我们发现在幕后还有更多事情：需要决定如何存储数据，并且需要变量和函数来操作这些数据。然而，所有这些变量和函数是这个类私有的(private)；客户程序不必知道它们是什么，它们是如何设计的或者对它们有任何访问。客户程序仅需要这个类说明和定义的公用(public)方法。

在本书中，我们将如下区分方法和函数，即使它们实际的语法(程序设计语法)是一样的：

约定

一个类的方法是公用的。

一个类中的函数是私有的。

1.2.4　Life:主程序

前面给出的 Life 游戏的算法大纲可转化成下面的C++程序。

```
# include "utility.h"
# include "life.h"
int main()                          //  Program to play Conway's game of Life.
/*Pre:  The user supplies an initial configuration of living cells.
  Post: The program prints a sequence of pictures showing the changes in the config-
        uration of living cells according to the rules for the game of Life.
  Uses: The class Life and its methods initialize(), print(), and update().
        The functions instructions(), user_says_yes(). */
{
  Life configuration;
  instructions();
  configuration.initialize();
  configuration.print();
  cout<<"Continue viewing new generations?"<< endl;
   while (user_says_yes()) {
     configuration.update();
     configuration.print();
     cout <<"Continue viewing new generations?"<< endl;
   }
}
```

* * * 实用程序包

此程序以包含文件开始,使得它能够使用类 Life 和标准C++的输入和输出库。实用函数 user_says_yes()在 utility. h 中声明,我们稍后讨论它。对于 Life 程序,我们所需要的关于 utility. h 的仅有的其他信息是它以如下的指令开始:

```
# include <iostream>
using namespace std;
```

这条指令允许我们使用标准的C++输入和输出流,如 cin 和 cout。(在较老的编译器上,一条替代的指令 # include <iostream. h>具有同样的效果。)

* * * 程序说明

Life 程序的文档从它的说明开始,即,程序开始时要求拥有的条件和它结束后将拥有的条件的准确说明。这些条件分别称为程序的前置条件和后置条件。为每个函数包含准确的前置条件和后置条件不仅清晰地解释了函数的目的,也有助于我们避免函数间的接口错误。由于包含说明非常有帮助,因此我们将它挑选出来作为我们的第一个程序设计技术规则:

程序设计技术规则

编写的每个程序、函数和方法要包含准确的前置条件和后置条件。

* * * 函数

程序说明的第三部分是它所使用的类和函数的列表,每个程序、函数或方法都应包含一个类似的列表。

棋跳盘上创建这个 Life 配置，用一种颜色的棋子代表当前一代中活的单元，并用第二种颜色的棋子标记那些将在下一代出生或死去的单元。）

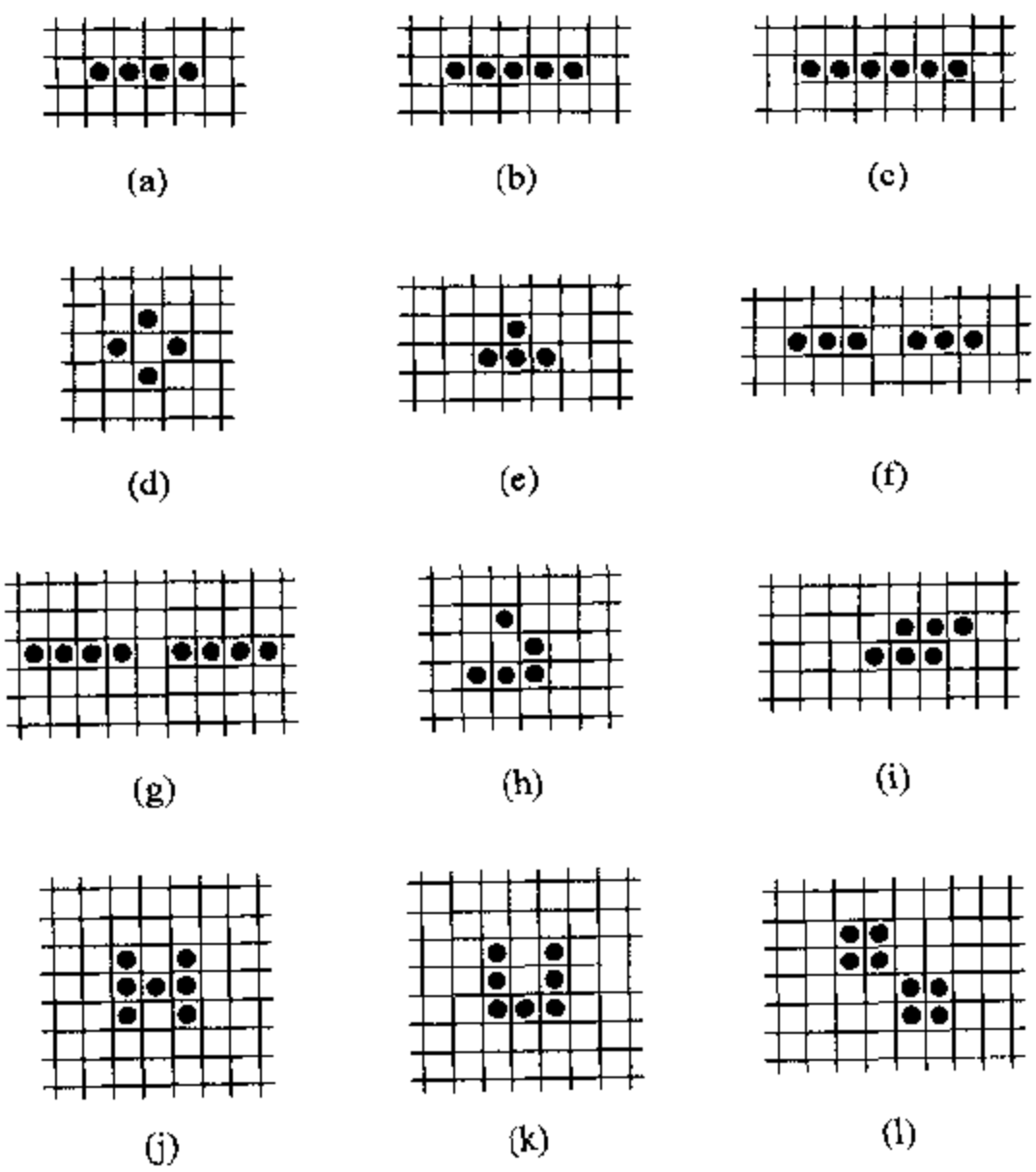


图 1.1　简单的 Life 配置

1.3　程序设计风格

在着手实现 Life 游戏的类和函数前，我们先暂停一下，以考虑在程序设计中应该注意使用的几个原理。

1.3.1　名称

在创世故事中（《圣经》创世纪 2:19），上帝带着所有的动物来看亚当能给它们取什么名字。按照旧的犹太传统，只有当亚当命名一种动物后，这种动物才会有生命。这个故事给计算机程序设计提出了一条重要的道德标准：即使数据和算法事先已存在，只有当它们有了有意义的名称，并且它们在程序中的位置能被恰当地识别和理解，它们才能获得自己的生命。

*** 审慎命名的目的

为了使程序完善地运行，最重要的是确切地知道每个类和变量代表什么以及每个函数做什么，因此一般应该包括对类、变量和函数进行解释的文档。我们应该审慎地选择类、变量和函数的名称，以明确、简洁地识别它们的意义。找到好名称通常并非一件易事，但它却

足够重要，我们将它挑出来作为第二个程序设计技术规则：

程序设计技术规则

最审慎地选择类、变量和函数的名称，并予以详尽的解释。

C++在这条技术规则的实施方面有相当的进展，它要求变量声明，而且给予我们几乎无限制的自由来标识名称。在不同的地方使用的常量应赋予名称，不同的数据类型也应如此，所以编译器能够捕获那些本来难以发现的错误。

我们会发现类型和类在C++程序中起着根本的作用，并且特别重要的是，它们应该被突出呈现给程序的阅读者。因此我们采用一种大写约定，这一点已在 Life 程序中应用：我们对任何类或程序设计员定义的类型标识符的起始字母使用大写。相反，对函数、变量或常量的标识符则使用小写字母。

名称的审慎选择在阐明程序和避免打印与常见错误方面大有帮助，指导原则如下：

* * * 指导原则

(1) 特别审慎地选择在程序的不同部分中使用的类、函数、常量和所有全局变量的名称。这些名称应该是有意义的，并且应该能够明确地指示这个类、函数、变量等的目的。

(2) 对仅是暂时、局部使用的变量，保持其名称简单。数学家通常使用单个字母代表一个变量，因此有时候当我们编写数学程序时，有可能对数学变量使用单个字母的名称。然而，即使是对控制 for 循环的变量，也经常有可能找到一个比较短但却较有意义的单词，它能更好地描述此变量的用途。

(3) 使用通用的前缀或后缀来关联同一常规类别的名称。例如，在一个程序中使用的文件可以被称为：

```
input_file    transaction_file    total_file    out_file    reject_file
```

(4) 避免采用故意的误拼和无意义的后缀来获得不同的名称，在下列所有名称中

```
index    indx    ndex    index    index2    index3
```

只有一个(第一个)应被正常使用。当你试图引入这种类型的多个名称时，用它作为一个警示，告诉你自己应该再仔细想想并设计出更能描述其使用意图的名称。

(5) 避免选择那些本身意义与问题毫无关系或只有很少关系的漂亮的名称。下面的语句可能非常有趣，但它们是低劣的程序设计！

```
do {
  study();
} while (TV.in_hock());
if (! sleepy) play();
else nap();
```

(6) 避免选择拼写互相接近或者其他方面易于混淆的名称。

(7) 应小心使用字母“l”(小写的 ell)、“O”(大写的 oh)和“0”(零)。在词或数中，通常可以根据上下文区分出它们而不会带来问题，但“l”和“O”决不应被单独用作名称。考虑下面的例子：

```
l=1;    x=1;    x=1;    x-0;    O=0
```

1.3.2　文档和格式

＊＊＊文档的目的

最初大多数学生认为文档是程序结束后必须忍受的繁琐之事，只是保证标注者和教师能阅读它，因而不会因为含糊而损失学分。小型程序的作者确实能在头脑中记住所有的细节，因此仅需要文档来向其他人解释这个程序。对于大型程序（及数月过后的小型程序），不可能记住每个细节如何与其他细节相关，因此为了编写大型程序，与程序的每个较小部分一起准备合适的文档是必要的。一种良好的习惯是在编写程序的同时准备文档，但如同我们随后将看到的，一种更好的习惯是在开始书写程序之前就准备部分文档。

并非所有的文档都是合适的。与很少有文档或仅有含义模糊的注释的程序一样常见的是那些带有冗长文档的程序，但这些文档却很少有助于增加对程序的理解。因此我们的第三个程序设计技术规则是：

程序设计技术规则

保持文档简练但具有描述作用。

文档的风格同所有的书写风格一样，是高度个人化的，许多不同的风格都证明是有效的。尽管如此，还是应当遵守一些普遍接受的指导原则：

＊＊＊指导原则

1. 在每个函数的开始处放上序言，包括：
 (1) 身份证明（程序设计员的名称、日期和版本号）。①
 (2) 所用函数和算法的目的的说明。
 (3) 函数所做的修改及其所使用的数据。
 (4) 对程序外部更多文档的引用。
2. 当定义每个变量、常量或类时，解释清楚它是什么及如何使用。如果从名称上就能明显看出这些信息则更好。
3. 对程序的每个重要的片断（段落或函数），用一句注释简要说明它的目的或动作。
4. 如果每个重要片断的结束不明显，则加以指示。
5. 避免机械模仿代码功能的注释，如：

   ```
   count++;      // 将 counter 加 1。
   ```

 或者使用无意义的行话，如

   ```
   // Horse string length into correctitude。
   ```

 （这个例子直接取自一个系统程序。）
6. 对任何使用了技巧或意义不清楚的语句加以解释，如果能避免使用这样的语句则更好。

① 为节省空间，本书中打印的程序不包含身份证明行或者序言的一些其他部分，因为周围的文本已经给出了必要的信息。

7. 代码本身应解释程序是如何工作的。文档应解释它为什么工作及它做什么。
8. 无论何时修改一个程序，确信文档得到了相应的修改。

* * * 格式

程序中的空格、空白行和缩排是一种重要的文档形式。它们使程序易于阅读，使我们能一眼看出程序的哪些部分彼此相关、哪里出现了主要中断以及每个循环中或者条件语句的每个选择分支当中分别包含哪些语句。有许多系统（有些是自动的）可用于缩排和调整空格，它们的目的都是为了更容易地确定程序的结构。

* * * prettyprinting

prettyprinter 是一个系统实用程序，它读入一个C++程序，在行间移动文本并调整空行，以此改进程序的外观并使其结构更清晰。如果在你的系统中有 prettyprinter 的话，你可以用它做试验，看看它是否有助于改进程序的外观。

* * * 一致性

由于程序的良好格式的重要性，需要为空格和缩排设置一些合理的规则并在所书写的所有程序中一致地使用这些规则。如果要使格式化系统在阅读程序中有用，则一致性是必要的。许多专业程序设计团体会选定一种统一的体系并坚持他们所编写的所有程序都符合它。一些班级或学生程序设计小组也同样这样做。这样，一个程序设计员就更容易阅读和理解别的程序设计员的作品。

程序设计技术规则

阅读程序的时间比编写程序的时间多得多。

使阅读更容易。

1.3.3 细化和模块化

* * * 问题求解

* * * 再分

计算机不求解问题，而人会求解。通常求解过程的最重要的部分是将问题划分成更小的问题，这些小问题能够被更详细地理解。如果这些问题仍然太困难，则再次细分它们，如此等等。在任何大型组织中，最高管理部门不必担心每个活动的每个细节，他们必须将精力集中在总体目标和问题上，而将具体的责任委派给下属。另一方面，中层管理者不必处理所有事情；他们必须再次划分工作并将它交给其他人。计算机程序设计亦是如此。即使当项目小得用一个人就能自始至终完成它，对这个工作进行划分还是最重要的。从对问题的全面理解开始，将它再分成子问题，然后依次处理每个子问题而不用考虑其他子问题。

让我们用一句经典的格言来重新描述这个原理：

程序设计技术规则

不要只见树木不见森林。

＊＊＊自顶向下的细化

＊＊＊说明

这条原理称为自顶向下的细化，是编写能运行的大型程序的真正关键。这一原理隐含着详细的考虑可以延迟，但精确性和严格性并不能延迟。它不意味主程序变成某种模糊的、难以描述任务的实体。相反，主程序会将几乎所有的工作分派给各种类、数据结构和函数，并且当我们编写主程序时（这是我们首先要写的），我们确切地知道工作是如何划分的。然后在后面开发某个特定的类或函数时，我们就能在开始之前确切地知道它们将来做什么。

通常很难准确地确定如何将工作划分成类和函数，而且有时候曾经做出的决定必须在以后被修改。虽然如此，还是有一些指导原则能有助于决定如何划分一项工作：

程序设计技术规则

使用类来模拟程序的基本概念。

例如，我们的 Life 程序确实是处理 Life 游戏，因此创建一个类 Life 来模拟这个游戏。我们经常可以通过用文字描述任务并对所用的不同名词各指派一个类，从而挑选出应用程序中的重要的类，我们所使用的动词经常意味着重要的函数。

程序设计技术规则

每个函数应该仅仅完成一项任务，但要很好地完成。

也就是说，我们应该能够简洁地描述一个函数的目的。如果你发现自己写了很长的一段话去说明一个函数的前置条件或后置条件，那么或者你给出了太多的细节（即，你提前写出了函数），或者你应该重新考虑工作的划分。函数本身毫无疑问会包含许多细节，但这些细节应该直到下一阶段的细化时才出现。

程序设计技术规则

每个类或函数应该隐藏某些东西。

大型公司的中层管理部门不会将部门的所有信息都传递给领导，他们首先概括、比较并清除信息，自己处理其中的许多请求，然后仅将需要的东西上报给上级领导。类似地，管理者不会将来自更高层管理部门的所有东西都传达给自己的下属，他们仅向下属传达开展工作所需要的那些东西。我们所编写的类和函数也应该一样。换言之，我们应该实施信息隐藏。

＊＊＊参数

细化过程最重要的部分之一就是确切地确定每个函数的任务，精确地说明它的前置条件和后置条件。也就是说，它的输入是什么，而且它将产生的结果又是什么。这些说明中的错误是最常发生的错误故障之一，也是最难以查找的错误之一。首先，必须对函数中使用的参数进行精确地说明。这些数据有 3 种基本类型：

- 输入参数由函数使用但并不被函数修改。在C++中，输入参数通常是通过值传递。

（例外：大对象应通过引用传递[①]，这样避免了建立本地副本所需要的空间和时间。然而，当我们通过引用传递一个输入参数时，应该在它的前面用关键字 const 加以声明。类型修改符 const 的使用是很重要的，因为它让阅读者知道我们是在使用一个输入参数，并允许编译器检测对这个参数的意外修改，有时候也允许编译器优化我们的代码。）

- 输出参数包含函数的计算结果。在本书中，输出参数使用引用变量。相反，C 程序设计员需要通过传递变量地址来模拟引用变量以利用输出参数。当然，C 的方法在 C++ 中仍然可用，但我们应避免使用。
- 输入输出参数既用作输入，也用作输出，函数使用参数的初始值并随后修改它。我们应通过引用来传递输入输出参数。

* * * 变量

除了参数外，函数使用的其他数据对象通常属于下列类别之一：

- 局部变量在函数内部定义，仅当函数运行时存在。它们在函数开始前不会被初始化，并且当函数结束时就废弃。

* * * 副作用

- 全局变量用于函数中但并不在函数中定义。在函数中使用全局变量往往很危险，因为在函数编写完毕后，程序作者可能完全忘了曾用过哪些全局变量以及如何使用它们。如果主程序在以后被修改的话，则这个函数可能会神秘地变得行为异常。如果函数改变一个全局变量的值，这种情况会引起副作用。副作用比使用全局变量作为函数的输入更危险，因为副作用可能改变其他函数的性能，从而在程序员调试程序时，经常将他误导到程序中已经正确的部分。

程序设计技术规则

保持连接简单。尽可能避免使用全局变量。

程序设计技术规则

只要能够避免，切勿引起副作用。

如果必须使用全局变量作为输入，则详细地将它们写入文档。

尽管这些自顶向下的设计原理看起来几乎是不言而喻的，但全面学习它们的惟一方法是通过实践。因此贯穿本书我们都仔细地将它们应用到所编写的大型程序中，而且现在就适合返回去讨论我们的第一个例子项目。

练习 1.3

E1. 在实现下列项目中，适合定义什么类？每个类将拥有什么方法？

(1) 存储电话号码的程序。

① 参考C++课本中关于通过引用进行调用和引用变量的讨论。

(2) 玩垄断游戏的程序。

(3) 玩井字游戏的程序。

(4) 模拟在带有红绿灯的十字路口等待的汽车队列的程序。

E2. 下面的类应该模拟一幅扑克牌，重写它的定义，使它符合风格原理。

```
    class a {           //  a deck of cards
int X; thing Y1[52]; /* X is the location of the top card in the deck. Y1 lists the
cards. */ public: a();
void Shuffle();         //  Shuffle randomly arranges the cards.
thing d();              //  deals the top card off the deck
}
            ;
```

E3. 给定下面声明：

```
int a[n][n], i, j;
```

这里 n 是一个常量。判断下面的语句的功能，并且重写这些语句，用非技巧的方法达到同样的效果。

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++)
    a[i][j]=((i+1)/(j+1))* ((j+1)/(i+1));
```

E4. 重写下面的函数，用非技巧的办法实现同样的结果。

```
void does_something(int &first, int &second)
{
 first = second - first;
 second = second - first;
 first = second + first;
}
```

E5. 判断下列每个函数的功能。用有意义的变量名、较好的格式重写每个函数，并去掉其中不必要的变量和语句。

(1)
```
int calculate(int apple, int orange)
{ int peach, lemon;
peach=0; lemon=0; if (apple<orange)
peach=orange; else if (orange<=apple)
peach=apple; else { peach=17;
lemon=19; }
return (peach);
}
```

(2) 假设声明 **typedef float** vector [max];：

```
float figure (vector vector1)
{ int loop1, loop4; float loop2, loop3;
loop1=0; loop2=vector1[loop1]; loop3=0.0;
loop4=loop1; for (loop4=0;
loop4<max; loop4++) {loop1=loop1+1;
loop2=vector1[locp1-1];
loop3=loop2+loop3; }loop1=loop1-1;
```

```
    loop2=loop1;
    return (loop2=loop3/loop2);}
```

(3)
```
int question(int &a17, int &stuff)
{ int another, yetanother, stillonemore;
another=yetanother; stillonemore=a17;
yetanother=stuff; another=stillonemore;
a17 = yetanother; stillonemore=yetanother;
stuff=another; another=yetanother;
yetanother=stuff; }
```

(4)
```
int mystery(int apple, int orange, int peach)
{ if (apple>orange) if (apple>peach)if
(peach>orange)return (peach); else if (apple<orange)
return (apple);else return (orange);else return (apple); else
if (peach>apple) if (peach>orange)return (orange); else
return (peach); else return (apple); }
```

E6. 下面设计的语句是用来检查3个整数的相对大小,假设这3个整数彼此不同:

```
if (x<z)if (x<y)if (y<z)c=1; else c=2; else
  if (y<z)c=3; else c=4; else if (x<y)
  if (x<z)c=5; else c=6; else if (y<z)c=7; else
  if (z<x)if (z<y)c=8; else c=9; else c=10;
```

(1) 以易于阅读的格式重写这些语句。

(2) 由于3个整数仅有6种可能的顺序,因此实际上仅会发生10种情况中的6种。找出那些决不会发生的情况,并消除其中冗余的检查。

(3) 编写更简单、更简短的语句来实现同样的结果。

* * * 立方根

E7. 下面的C++函数计算一个浮点数的立方根(通过牛顿近似法),使用的原理是:如果 y 是 x 的立方根的一个近似值,则 $z=\dfrac{2y+x/y^2}{3}$ 是一个更接近的近似值。

```
float function fcn(float stuff)
{ float april, tim, tiny, shadow, tom, tam, square; int flag;
tim=stuff; tam=stuff; tiny=0.00001;
if (stuff != 0) do {shadow=tim+tim; square=tim* tim;
tom=(shadow+stuff/square); april=tom/3.0;
if (april*april*april-tam>-tiny) if (april*april*april-tam
    <tiny) flag=1; else flag=0; else flag=0;
if (flag==0)tim=april;else tim=tam;}while (flag!=1);
if (stuff==0) return (stuff); else return (april); }
```

(1) 用有意义的变量名重写这个函数,去掉其中无益于理解的多余的变量,使它有更好的布局,而且没有冗余的和无用的语句。

(2) 编写一个函数从数学公式直接计算出 x 的立方根,以赋值 $y=x$ 开始,然后重复 $y=(2*y+(x/(y*y)))/3$ 直到 $\text{abs}(y*y*y-x)<0.00001$。

(3) 这些任务中的哪一个更容易?

＊＊＊统计

E8. 一个数序列的均值是这个序列中的所有数的和除以数的个数。这个序列的(总体)方差是序列中所有数的平方的均值，再减去序列中的数的均值的平方。标准方差是方差的平方根。编写一个结构良好的C++函数来计算有 n 个浮点数的一个数序列的标准方差，这里 n 是一个常量并且这些浮点数在数组中的下标是从 0 到 $n-1$，n 和数组作为函数的参数。使用并随后编写辅助函数来计算均值和方差。

＊＊＊标图

E9. 设计一个程序在图上标绘出给定的点集。此程序的输入是一个文本文件，文件的每一行包含两个数，它们分别是要标绘的点的 x 和 y 坐标。此程序将使用一个函数来标绘一个这样的坐标对，这个函数的细节涉及到标绘的特定方法，由于它们依赖于标绘设备的要求，而我们并不知道这些要求，因而未能写出这个函数的细节。程序在标绘点之前需要知道其输入文件中 x 和 y 的最大值、最小值。因此程序要使用另一个函数 bounds 来读入整个文件并判断这 4 个最大值、最小值。其后，程序将使用另一个函数来画出并标注数轴，然后重置输入文件并标绘各个点。

(1) 编写主程序，不包括函数。

(2) 编写函数 bounds。

(3) 写出剩余函数的前置条件和后置条件，同时加上合适的文档，说明函数的目的和需求。

1.4　编码、测试和进一步细化

本节标题中的 3 个过程是联合并进的，必须一起进行。然而我们在思想上还是让它们保持独立，因为每个过程都需要自己的步骤和方法。当然，编码是用一门计算机语言如C++的正确语法(语法规则)编写算法的过程；测试是用所选的样本数据运行程序以发现所出现的错误的过程。在进一步细化过程中，我们着手处理尚未编写的函数并重复这些步骤。

1.4.1　占位函数

＊＊＊及早调试和测试

在主程序的编码完成后，大多数程序设计员希望尽早完成所需要的类和函数的书写和编码，以查看整个项目是否能正常工作。对于像 Life 游戏这样小的项目来说，这种方法可能是有效的。但对更大型的项目，书写和编码会是一项很艰巨的工作，当它完成的时候，在早期所写的主程序和类及函数的细节可能都被忘记了。实际上，可能是由不同的人编写不同的函数，而其中一些启动这个项目的人可能在所有函数编写完成前就离开了项目组。在脑子里比较新鲜的项目会更易于理解和调试。因此，对大型项目来说，当每个类和函数一编写完就进行调试和测试比等到项目全部编码完成后再进行调试和测试更加有效。

即使对较小的项目，也有理由对类和函数一次一个地进行调试。例如，我们可能不能确定将在程序中几处位置上出现的某个C++语法。如果能分别编译每个函数，则我们能很快地学会在后继函数的语法中避免类似错误。作为第二个例子，假定我们已确定程序的主要

步骤应按某种顺序进行，如果我们一编写完主程序就测试它，则可能会发现有时候主要的步骤是按错误的顺序进行的，那么我们就能很快地改正程序，这比等到主要步骤可能被它所包含的许多细节所遮掩的时候再来测试它要更加容易。

* * * 占位函数

为正确地编译主程序，在用到的每个函数的位置上应该有点东西，因此我们必须放上简短的哑函数，又称为占位函数。最简单的占位函数只做很少工作或完全不做任何事情：

```
void instructions(){ }
bool user_says_yes(){return(true);}
```

注意在编写占位函数时，我们至少必须约束它们的相关参数和返回类型。例如，在为user_says_yes()设计占位函数时，我们决定它应该返回一个简单的 true 或 false，这意味着我们应该将此函数的返回类型定为布尔型(bool)。类型 bool 只是在最近才被加到C++中，一些较老的编译器不识别它，但我们通常可以用下面的语句来模拟它——按照惯例，如果需要的话，可以将它放在实用程序包中：

```
typedef int bool;
const bool false=0;
const bool true=1;
```

除了占位函数以外，我们的程序中也需要类 Life 的占位定义。例如，在文件 life.h 中，我们应该如下定义这个类，不必带数据成员：

```
class Life{
public:
  void initialize();
  void print();
  void update();
};
```

在 life.c 中我们也要为这个类的方法提供下列占位函数：

```
void Life::initialize() {}
void Life::print() {}
void Life::update() {}
```

注意这些方法定义必须使用C++的作用域解析运算符 :: ①以指示它们属于类 Life 的作用域。

即使是用这些最小的占位函数，我们至少能编译程序并确信类型和变量的定义在语法上是正确的。然而，通常每个占位函数应该打印一条说明此函数被调用的信息。当我们执行程序时，我们发现它进入一个无限循环，因为函数 user_says_yes()总是返回 true 值。但是主程序毕竟编译通过而且能够运行了，那些我们可以继续细化这些占位函数。对于像 Life 游戏这样的小项目，我们完全可以依次编写每个类或函数，用它替换对应的占位，并观察程序执行的结果。

① 参考C++课本中对作用域解析运算符和类方法语法的讨论。

1.4.2　类 Life 的定义

* * * 1：活的单元

* * * 0：死的单元

每个 Life 对象需要包含一个矩形数组[①]，我们称之为 gird，用来存储 Life 配置。在数组 grid 中，我们用整数 1 表示活的单元，用 0 表示死的单元。因此为了对某个特定单元的活的邻居计数，只要累加其邻居单元的值即可。实际上，在更新 Life 配置时，我们要重复计算此配置中各个单元活的邻居的数目。因此，类 Life 应包含一个成员函数 neighbor_count 来完成这项任务。另外，由于客户程序的代码并不需要成员 neighbor_count，因此我们将这个函数的可见性定为私有(private)。相反，前面的 Life 函数都必须拥有公用(public)可见性。最后，必须决定 Life 配置所承载的矩形数组的维数。我们将这些维作为全局常量编码，因此进行一下简单的修改就可以按我们的需要重新设置程序中网格的大小。注意常量可以安全地放在.h 文件中。

```
const int maxrow=20, maxcol=60; //  grid dimensions

class Life {
public:
  void initialize();
  void print();
  void update();
private:
  int grid[maxrow+2][maxcol+2];
                          //  allows for two extra rows and columns
  int neighbor_count(int row, int col);
};
```

我们不用编写成员函数，通过一同使用前面的占位函数和这个私有函数 neighbor_count 的占位函数，就可以测试这个定义。

1.4.3　对邻居计数

* * * 函数 neighbor_count

* * * 栅栏

现在进一步细化我们的程序。对坐标为 row，col 的单元的邻居进行计数的函数要求查看它的 8 个邻接单元。我们需要使用一对 for 循环来完成它，一层循环从 row－1 运行到 row+1，另一层循环从 col－1 到 col+1。需要小心的是，当 row，col 在网格的边界上时，只查看网格中合法的单元。我们不是采用几个 if 语句来确认有没有跑出网格，而是在网格周围引入栅栏(hedge)：通过增加额外的两行，其中一行是在网格的第一个实行前，另一行在最后一个实行之后，也增加了额外的两列，其中一列是在网格的第一个实列之前，另一列是在最后一个实列之后，从而扩大了网格。在对类 Life 的定义中，我们事先已考虑到栅栏，将成员

① 具有两个下标的数组称为矩形，第一个下标确定数组中的行，第二个确定列。

grid 定义为一个具有 maxrow+2 行和 maxcol+2 列的数组。栅栏行和列中的单元始终是死的，因此完全不会影响活的邻居的计数。但它们的出现意味着对邻居计数的 for 循环不用再区分网格边界上的行或列与其他的行或列。参见图 1.2 中的例子。

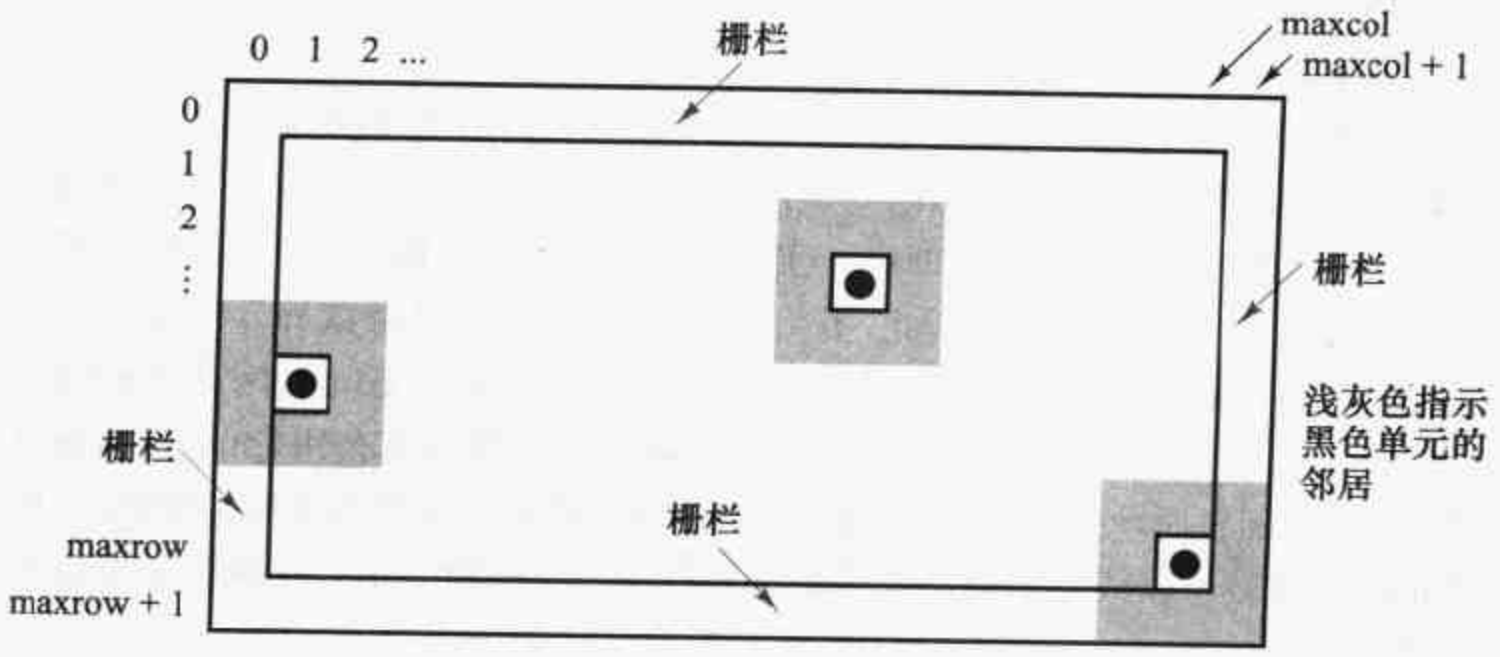


图 1.2 带栅栏的 Life 网格

* * * 监视哨

另一个经常使用的替代栅栏的术语是监视哨(sentinel)：监视哨是放入数据结构中的额外的元素，因而不用再将边界条件作为一种特殊情况对待。

```
int Life::neighbor_count(int row, int col)
/*Pre:  The Life object contains a configuration, and the coordinates row and col
        define a cell inside its hedge.
 Post: The number of living neighbors of the specified cell is returned. */
{
  int i, j;
  int count=0;
  for (i=row-1; i<=row+1; i++)
    for (j=col-1; j<=col+1; j++)
      count+=grid[i][j];  //  Increase the count if neighbor is alive.
  count-=grid[row][col];
                          //  Reduce count, since cell is not its own neighbor.
  return count;
}
```

1.4.4 更新网格

* * * 方法 update

用来更新 Life 配置的方法的动作简单明了。首先使用存储在配置中的数据来计算名为 new_grid 的矩形数组的各个元素，new_grid 记录了更新后的配置。然后逐个元素地将 new_grid 复制回 Life 对象的 grid 成员中。

为创建 new_grid，我们用一个在行 row 和列 col 上的嵌套的循环，在矩形数组 grid 中的非栅栏元素上运行一遍。这些嵌套的循环体由多路选择语句 switch 组成。函数 neighbor_

count(row, col)的返回值是 0，1，…，8 中的某一个，对其中的每种情况，可以采取一种不同的动作，或者就像我们的应用程序中，几种情况可能导致同一种动作。应当检查每种情况中所规定的动作正确对应到了 1.2.1 节中的规则 2，3，4，5。

```
void Life::update()
/*Pre: The Life object contains a configuration.
  Post: The Life object contains the next generation of configuration. */
{
  int row, col;
  int new_grid[maxrow+2][maxcol+2];
  for (row=1; row<=maxrow; row++)
    for (col=1; col<=maxcol; col++)
      switch (neighbor_count(row, col)) {
      case 2:
        new_grid[row][col] = grid[row][col]; //  Status stays the same.
        break;
      case 3:
        new_grid[row][col]=1;   //  Cell is now alive.
        break;
      default:
        new_grid[row][col]=0;   //  Cell is now dead.
      }
  for (row=1; row<=maxrow; row++)
    for (col=1; col<=maxcol; col++)
      grid[row][col]=new_grid[row][col];
}
```

1.4.5　输入和输出

＊＊＊审慎地输入和输出

现在用函数 user_says_yes()和 instructions()来作为程序的输入和输出后，仅剩下编写类 Life 的 initialize()和 print()方法。在设计供许多人使用的计算机程序中，执行输入和输出的程序经常是最长的。程序中必须对输入进行充分检查，确信它是有效的和一致的，并且必须处理输入中的错误以避免灾难性的失败或产生荒谬的结果。程序中也必须仔细地组织输出和编排输出格式，充分考虑什么内容应该打印，什么内容不应该打印，要准备各种选择以满足不同的情况。

程序设计技术规则

将输入和输出作为独立的函数，使得它们易于修改并能定制修改以适应计算系统。

1. instructions

＊＊＊流输出运算符

instructions()函数是对插入运算符 <<和标准输出流 cout 的一个简单练习。注意到我们采用控制符 endl 来结束一行并刷新输出缓冲区。在仅希望刷新输出缓冲区而不结束一行的地方，我们可以替代地使用控制符 flush。关于C++中流输入和输出的准确描述，请参

考C++的课本。

```
void instructions()
/*Pre: None.
  Post: Instructions for using the Life program have been printed. */
{
  cout<<"Welcome to Conway's game of Life."<<endl;
  cout<<"This game uses a grid of size"
      <<maxrow<<"by"<< maxcol<<"in which"<<endl;
  cout<<"each cell can either be occupied by an organism or not."<<endl;
  cout<<"The occupied cells change from generation to generation"
      <<endl;
  cout<<"according to the number of neighboring cells which are alive."
      <<endl;
}
```

2. initialization

* * * 输入方法

Life 的 initialization()方法必须完成的任务是设置初始配置。为初始化 Life 对象，我们可以分别考虑每个可能的坐标对并要求用户指出此单元是否被占据。这种方法要求用户输入 maxrow * maxcol=20 * 60=1200 个记录项，这个数太大了。因此，我们仅输入对应于最初被占据单元的那些坐标对。

```
void Life::initialize()
/*Pre: None.
  Post: The Life object contains a configuration specified by the user. */
{
  int row, col;
  for (row=0; row<=maxrow+1; row++)
    for (col=0; col<=maxcol+1; col++)
      grid [row][col]=0;
  cout<<"List the coordinates for living cells."<<endl;
  cout<<"Terminate the list with the the special pair-1-1"<<endl;
  cin>>row>>col;
  while (row!=-1 || col!=-1){
    if (row>=1 && row<=maxrow)
      if (col>=1 && col<=maxcol)
        grid[row][col]=1;
      else
        cout<<"Column"<<col<<"is out of range."<<endl;
    else
      cout<<"Row"<<row<<"is out of range."<<endl;
    cin>>row>>col;
  }
}
```

* * * 输出

对输出方法 print()，我们采用简单的方法，输出每一代的整个矩形数组，其中被占据的单元用 * 表示，空单元用空格表示。

```
void Life::print()
```

```
/*Pre: The Life object contains a configuration.
  Post: The configuration is written for the user. */
{
  int row, col;
  cout<<"\nThe current Life configuration is:"<<endl;
  for (row=1; row<=maxrow; row++) {
    for (col=1; col<=maxcol; col++)
      if (grid[row][col]==1)cout<<'*';
      else cout<<' ';
    cout<<endl;
  }
  cout<<endl;
}
```

*** 用户响应**

最后来看函数 user_says_yes(),它判断用户是否要继续计算下一代。user_says_yes()的任务是请求用户响应"是"或者"否"。为使程序对输入具有更强的容错能力,我们将此请求放在一个循环中,它一直重复直到用户的响应可以接受。在我们的函数中,我们使用标准的输入函数 get()来每次处理一个输入字符。在C++中,函数 get()实际上仅是类 istream 的一个方法:在我们的应用程序中,应用的方法 cin. get()属于 istream 的对象 cin。

```
bool user_says_yes()
{
  int c;
  bool initial_response=true;
  do {                          //  Loop until an appropriate input is received.
    if (initial response)
      cout<<" (y,n)?"<<flush;
    else
      cout<<"Respond with either y or n:"<<flush;
    do {                        //  Ignore white space.
      c=cin.get();
    } while (c=='\n'||c==' '||c=='\t');
    initial_response=false;
  } while (c!='y' && c !='Y' && c!='n' && c!='N');
  return (c=='y' || c=='Y');
}
```

至此,我们已经有了供模拟 Life 用的所有函数,现在是暂停一下并检查它是否能正常工作的时候了。

1.4.6 驱动器

*** 个别调试**

就小型项目来说,通常在每个函数写完后就将它插入到程序中合适的位置,使最后得到的程序随后能尽可能地被调试和测试。然而,对大型项目来说,整个项目的编译会掩盖正被调试的新函数,仅通过运行整个程序的方式,很难辨别某个特定函数是否正确运行。即使是在小型项目中,一个函数的输出也可能被另一个函数使用,但另一个函数在使用这个输出时却不能立即知道所传送的信息是否正确。

* * * 驱动程序

调试和测试单个函数的一种方法是写一个较短的辅助程序来为此函数提供必要的输入、调用此函数并评价函数执行的结果.这样的辅助函数称为函数的驱动器。通过使用驱动器,可以将每个函数分离出来逐个研究,因此经常能很快地发现错误。

作为一个例子,我们为 Life 项目的函数编写驱动器。首先考虑 neighbor_count()方法。在我们的程序中,使用了这个函数的输出,但并未将它直接显示出来以供检查,因此我们对这个函数是否正确毫无信心。为了测试 neighbor_count(),需要提供 Life 的对象 configuration,对 configuration 的每个单元调用 neighbor_count,然后输出结果。最后形成的驱动器使用 configuration. initialize()来设置对象,而且与原来的主程序有几分相似。为了从驱动器中调用 neighbor_count(),需要在类 Life 中将函数的可见性暂时改为 public。

```
int main ()                    //driver for neighbor count()
/*Pre: None.
  Post: Verifies that the method neighbor_count() returns the correct values.
  Uses: The class Life and its method initialize(). */
{
  Life configuration;
  configuration.initialize();
  for (row=1; row<=maxrow; row++){
    for (col=1; col<=maxrow; col++)
      cout<<configuration.neighbor count(row, col)<<" ";
    cout<<endl;
  }
}
```

有时候两个函数可以用于互相检查。例如,检查 Life 的方法 initialize()和 print()的最简单的方法是用一个驱动器,这个驱动器的动作部分是:

```
configuration.initialize();
configuration.print();
```

通过运行这个驱动器并确信打印出来的配置与作为输入的配置是一样的,可以同时测试这两个方法。

1.4.7 程序跟踪

* * * 小组讨论

在将函数组装成完整的程序后,就可以检查整个程序了。发现潜在缺陷最有效的方法之一称为结构化预排(structured walkthrough)。在预排时,程序设计员向其他程序设计员或一个程序设计员小组展示所完成的程序并准确地加以解释:首先解释主程序,接下来逐个介绍各个函数。有 3 个原因表明结构化预排是有用的:首先,对实际代码并不熟悉的程序设计员经常能发现原程序设计员所忽略的 bug 和概念上的错误。其次,别人所问的问题有助于澄清思想并发现自己的错误。最后,结构化预排经常会提议一些测试思想,这些测试在以后的软件产品阶段往往证明是有用的。

* * * 调试用打印语句

大型程序首次作为整体执行就能正确运行的情况是比较罕见的,如果它未正确运行,则

要准确判断错误所在并不容易。许多系统上都有复杂的跟踪工具可以用来跟踪函数调用和变量修改等。然而，一种简单而有效的调试工具是通过在主程序的关键位置插入打印语句而得到程序执行的快照(snapshot)，这种策略也经常作为可用调试器的一个选项。可以在函数每次被调用时都打印一条信息，也可以在函数调用之前或调用之后打印出重要变量的值。这种快照能帮助程序设计者快速聚焦于错误发生的特定位置。

＊＊＊临时支架

支架(scaffolding)是另一个术语，经常用来描述为辅助调试而插入到一个程序中的代码。在书写程序时不要犹豫向程序中放入支架，当不再需要时删除它们也很简单，而且在调试过程它可能会使我们避免失败。

当程序中出现了某个完全无法定位的神秘的错误时，向主程序中放入支架打印出重要变量的值是非常有用的方法。应该将支架放在主程序中一或两个主要划分点的位置上，(如果你写了一个很长的程序而没有将它的工作再分为几个主要部分，则你已经在程序设计结构中犯下了严重的错误，因而必须加以改正。)利用主要划分点的打印输出，就能够判断程序的哪一部分行为异常。然后我们就可以将精力放在那个部分，在它的子划分中再引入支架。

＊＊＊防御式程序设计

另一种重要的检错方法是实施防御式程序设计。在函数开始处放上 if 语句检查是否确实满足前置条件，如果不满足，则打印一条错误信息。这样，当想像中的不会发生的情况确实出现时，就可以尽早得到警告。而如果不出现这种情况，则此处的错误检查对用户来说是完全不可见的。当然，当函数的输入是来自用户、文件或者程序外部的其他来源时，特别重要的是要检查前置条件能否满足。然而令人惊奇的是，无论多么经常地检查前置条件，也往往会发现有错，甚至是在那些已确信一切正常的地方发现错误。

＊＊＊静态分析器

对大型程序有时还会用到另一个工具，它就是静态分析器。静态分析器对源程序(例如用C++编写)进行检查以发现其中未曾初始化或未被使用过的变量、决不可能到达的代码段或者其他的可能错误的事故。

1.4.8　程序测试原理

＊＊＊选择测试数据

迄今为止，我们尚未讨论如何选择用于测试程序和函数的数据。当然，这种选择密切依赖于正在开发的项目，因此我们只能给出一些一般说明。首先注意以下技术规则：

程序设计技术规则

测试数据的质量比数量更重要。

多次用同样的案例做同样的计算而进行的抽样运行，其测试结果并不比一次运行更加有效。

程序设计技术规则

程序测试可用于说明 bug 的存在，而不能说明其不存在。

有可能在多次抽样运行后仍然还有从未被测试到的其他案例。对任何真正复杂的程序，做穷尽测试是不可能的，而测试数据的精心选择则可以提供对程序的实质信心。例如，每个人对典型的计算机能够正确地做两个浮点数的加法抱有极大的信心，但这种信心当然并非基于用所有可能的浮点数做加法来测试计算机。如果一个双精度浮点数占 64 位，则共有 2^{128} 个不同的数对要相加。这是一个天文数字：迄今为止制造的所有计算机一同工作，也只能执行其中很小的一部分加法。我们关于计算机正确做加法的信心是基于分别测试每种成分，即，通过检查每个 64 位数对能被正确地相加，并且从一处搬到另一处时也能被正确地相加。

* * * 测试方法

至少有 3 种通用的思想可以用于选择测试数据：

1. 黑盒方法

大型程序的多数用户对程序的功能细节并不感兴趣，他们仅希望得到答案，也就是说，他们希望将此程序作为一个黑盒(Black-box)对待。类似地，测试数据是用来检查程序能否正确运行，因此在选择测试数据时，应该按照问题的说明，而不用关心程序的内部细节。测试数据至少要用下列方式选择：

* * * 数据选择

(1)简易值。应当用易于检查的数据来调试程序。许多学生只用复杂的数据测试程序，而当教师尝试简单的样例时，虽然程序能完善地运行，但仍然会使这些学生感到局促不安。

(2)典型的、逼真的值。总是选择那些能展示如何使用此程序的数据来测试程序。这些数据应该足够简单，因而能够对结果进行手工检查。

(3)极值。许多程序在其应用范围的界限处出错。计数器或数组边界就很容易会被突破。

(4)非法值。“废料入，废料出”是计算机行业中一句不应被注意的古老谚语。当一个良好的程序用废料输入时，其输出至少应当是一句敏感的错误信息。实际上，程序应该对输入中可能的错误提供一些指示，并且在忽略错误的输入后仍可能执行一些计算。

2. 玻璃盒方法

* * * 路径测试

选择测试数据的第二种方法是以下面的观察为出发点，即如果一个程序实际上有某些部分的代码未被执行到，则不能认为此程序经过了全面的测试。在玻璃盒测试方法(The Glass-Box Method)中，要检查程序的逻辑结构，对其中出现的每个分支，都要设计测试数据通往那个分支。因此要仔细选择数据来检查每个 switch 语句中的每种可能、每个 if 语句中的每个分句以及每个循环的终止条件。如果程序中有数个选择或迭代语句，则要求使用不

同组合的测试数据来检查所有可能的路径。图 1.3 说明了一个短程序段的可能的执行路径。

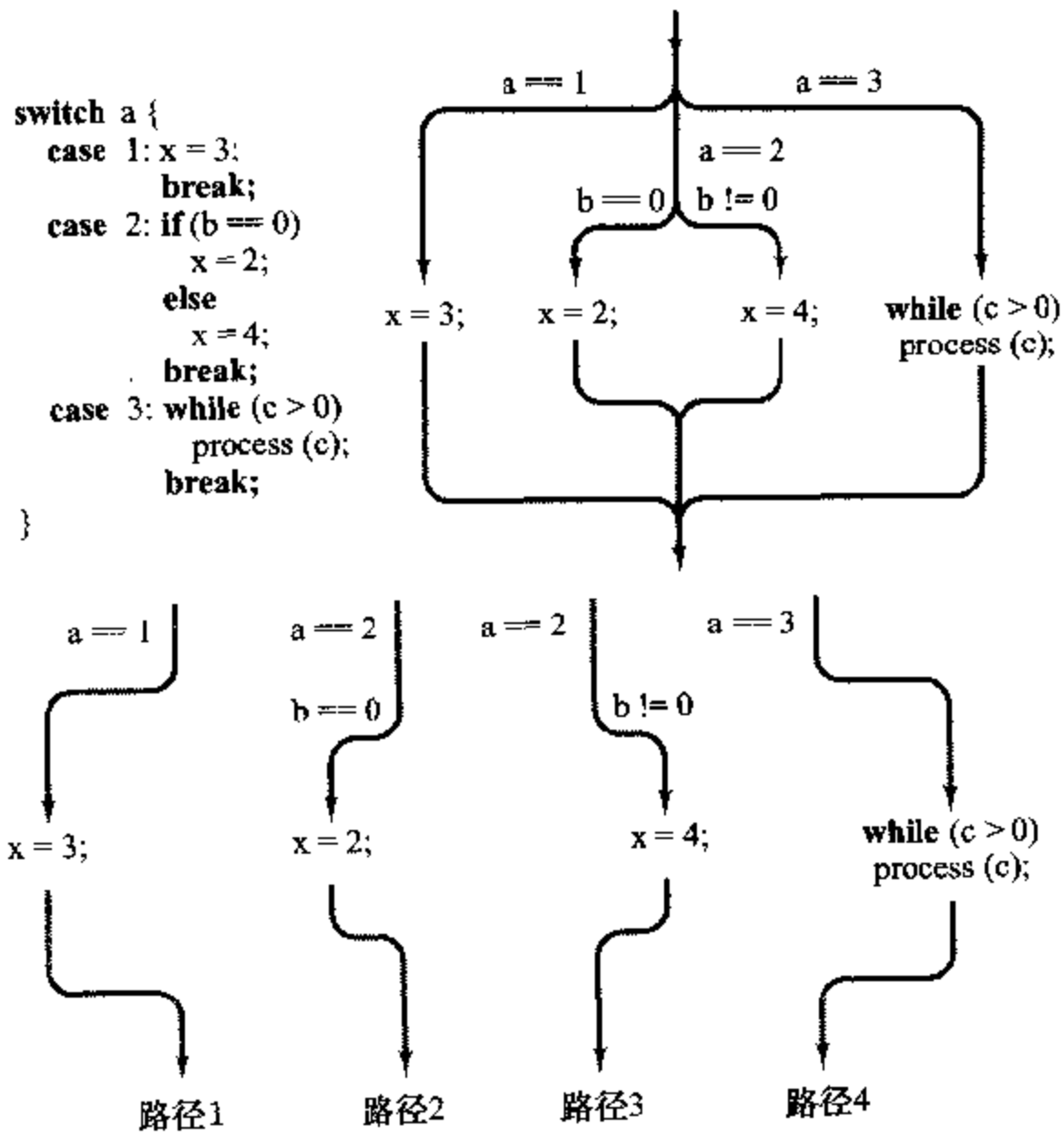


图 1.3　贯穿一个程序段的执行路径

对大型程序，玻璃盒方法显然是不可行的，但对单个的小模块，它是一种极好的调试和测试手段。在一个设计良好的程序中，每个模块都包含少数循环和分支，因此仅用一些精心选择的测试案例就足以独立地测试每个模块。

＊＊＊模块测试

在玻璃盒测试中，模块化程序设计的好处就表现得很明显。我们考虑一个包含 50 个函数的项目的典型例子，每个函数包含 5 个不同的情况或分支。如果我们打算将整个程序合一测试，则需要 5^{50} 个测试案例才能确认每个分支都被测试到。而每个模块分别只需要 5 个(更简单的)测试案例，总共是 $5\times 50=250$ 个。因此对大型程序，玻璃盒测试就将规模从难以承受缩减成相当适度。

＊＊＊比较

＊＊＊接口错误

在你得出玻璃盒测试总是更可取的方法这一结论之前，我们将说明，实践中，黑盒测试通常在发现错误方面更加有效。或许一个原因就是大多数微妙的程序设计错误经常不是发生在一个函数内部，而是发生在函数间的接口中，误解了函数间信息交换的准确条件和标准。因此对大型项目来说，一条合理的测试思想就是在编写每个小模块时就对它应用玻璃盒方法进行测试，而当程序完成时就使用黑盒测试数据来测试较大的部分。

3. Ticking-Box 方法

为了总结本节，让我们再说明一条程序测试思想。遗憾的是，这一思想的使用相当广泛，它或许能够被称为 ticking-box 方法。此方法在项目完成了相当完善的调试后完全不再做任何测试，取而代之的是将程序移交给客户来试用和认可。当然，其结果是一颗定时炸弹。

练习 1.4

E1. 如果你怀疑 Life 程序中包含错误，主程序中的何处是插入支架的较好位置？应该打印出什么信息？

E2. 提出对 1.3 节中练习 E9(设计程序来标绘…个点集)的解决方案，并指出在需要的情况下插入支架的较好位置。

E3. 为下面每个函数找出合适的黑盒测试数据：

(1) 此函数返回 3 个参数中的最大者，其参数为浮点数。

(2) 此函数返回一个浮点数的平方根。

(3) 此函数返回其两个参数的最小公倍数，参数必须是正整数。(最小公倍数是两个参数最小的公共整数倍数，例如：4 和 6 的最小公倍数是 12，3 和 9 的最小公倍数是 9，5 和 7 的最小公倍数是 35。)

(4) 此函数将 3 个作为参数给出的整数按升序排序。

(5) 此函数将下标从 0 到变量 n 的整数数组 a 按升序排序，这里 a 和 n 均为参数。

E4. 为下面每小题找出合适的玻璃盒测试数据：

(1) 语句

```
if (a<b) if (c<d) x=1; else if (c==d) x=2;
else x=3; else if (a==b) x=4; else if (c==d) x=5;
else x=6;
```

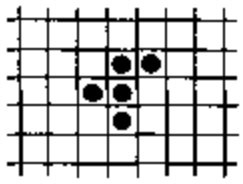
(2) Life 的方法 neighbor_count(row, col)。

程序设计项目 1.4

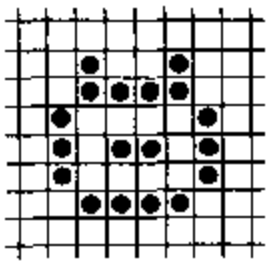
P1. 在计算机上输入本章的 Life 程序并确信它能正确运行。

P2. 用图 1.1 中所示的例子测试 Life 程序。

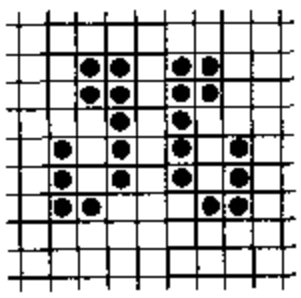
P3. 用图 1.4 中所示的初始配置运行 Life 程序，它们中有几个需要经历许多次变化，才能达到保持同样配置或者具有可预测的行为。



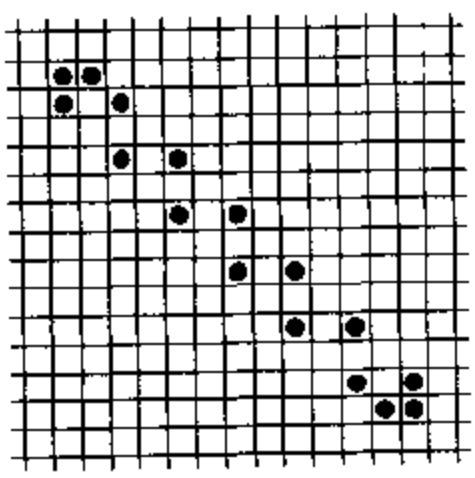
五格拼板



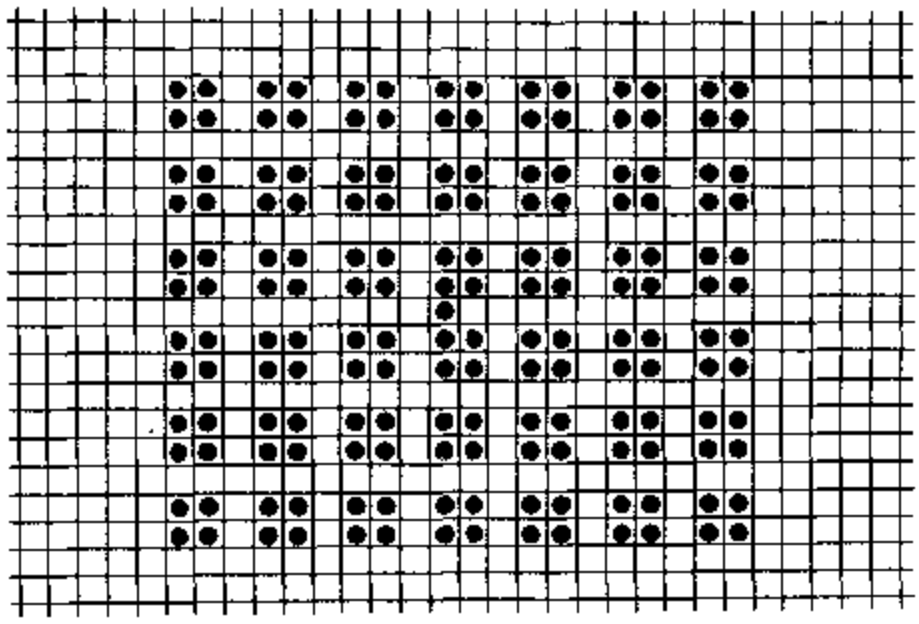
露齿嘻笑的猫



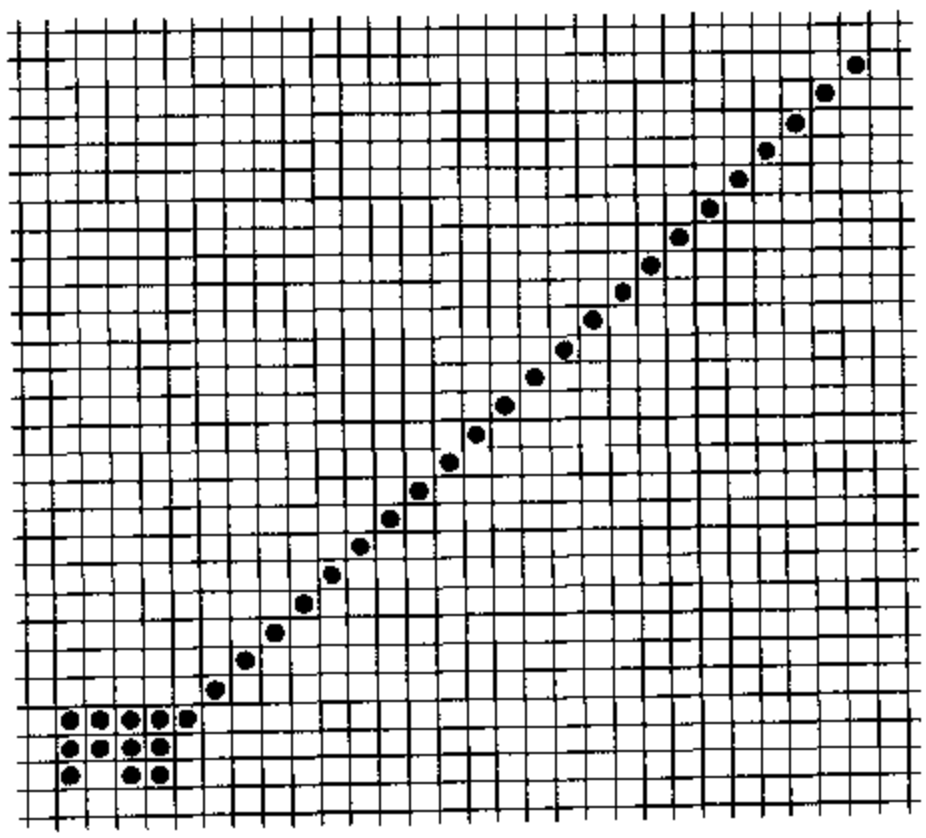
翻转



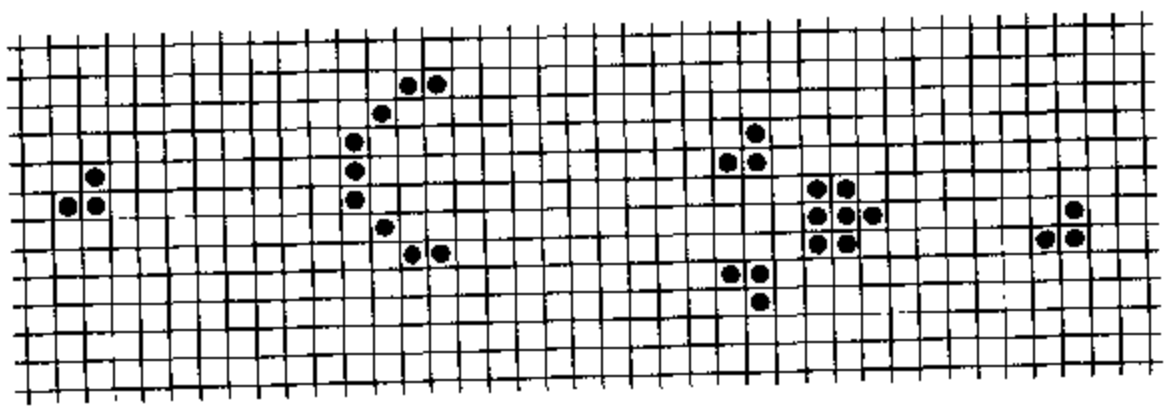
螺旋条状纹



病毒



收割机



滑膛枪

图 1.4　Life 配置

1.5　程序维护

作为练习或示范而编写的小程序往往在运行几次后就被丢弃了，但大型程序的保存就

完全不同了。有实际价值的程序通常会出许多不同的人运行许多次，而程序的编写和调试仅仅标志着使用的开始，同时也仅仅标志着使程序有用并保持它一直有用的工作的开始。因此有必要对程序进行复查和分析，以确保它满足指定的要求，使它能够适应不断变化的环境，并对它进行修改以使它更好地满足用户的需要。

计算机程序的维护(maintenance)包含在程序完全调试、测试并投入使用后对程序所做的所有工作。随着时间和经验的积累，人们对这个计算机程序的期望通常会发生变化。运行和硬件环境会发生变化，用户的需求和期望会发生变化，与软件系统其他部分的接口也会发生变化。因此，如果要使程序继续有用，那么必须对它保持注意力，使它保持最新。实际上，有调查表明：

程序设计技术规则

对一个大型且重要的程序，超过一半的工作是在
它已被完全调试、测试并投入使用后，来自于维护阶段。

1.5.1 程序评价

程序维护的第一步是开始复查、分析和评价这一连续的过程。我们可以对任何程序提出几个有用的问题，其中第一组问题涉及程序的使用和输出(因此是黑盒测试工作的继续)。

1. 程序是否解决了所要求的问题？是否准确地遵循了问题说明？
2. 程序在所有条件下都能正确工作吗？
3. 程序是否有良好的用户接口？是否能让用户方便简单地进行输入？程序的输出是否清晰、有用且界面漂亮？是否提供替代选择或可选特性以方便使用？是否为用户包含了明确且足够的指令和其他信息？

 其余的问题涉及程序的结构(是玻璃盒测试过程的继续)。
4. 程序的编写是否清晰且符合逻辑？是否有适于完成逻辑任务的方便的类和短函数？类中的数据结构是否正确地反映了程序的需要？
5. 程序是否有良好的文档？变量、函数、类型和方法的名称是否正确地反映了它们的意义？是否合适地给出了精确的前置条件和后置条件？是否对代码中的主要部分或任何特殊或困难的代码给出了解释？
6. 程序是否有效地利用了时间和空间？通过修改潜在的算法，程序的性能能否得到改进？

我们将对所编写的程序详细地研究其中的一些标准，对其他的准则不予明确地说明，但这并不表明它们不重要。而如果在程序设计的每一阶段都投入足够的思考和努力的话，这些标准能够自动被满足。我们希望所研究的例子能揭示这个道理。

1.5.2 复查 Life 程序

让我们通过重新考虑 Life 游戏的程序来详细说明这些程序评价标准。这样做在一定程度上实在是矫枉过正，因为像 Life 游戏这样的一个玩具项目本身并不值得这种努力。然而，

在此过程中，我们应该考虑对许多其他应用来说都十分重要的程序设计方法。我们将依次考虑前述的每个问题。

1. 问题说明

＊＊＊问题：边界

如果我们返回去复查 1.2.1 节中 Life 游戏的规则，就会发现实际上我们并未像最初所描述的那样解决 Life 游戏。规则中并未提到包含单元的网格的边界，而在我们的程序中，当一个移动的群体充分靠近边界时，则邻居的空间就消失了，由于边界的出现而使得此群体产生失真。这一点并不是规则所假定的，因此我们的程序违反了规则。

在任何计算机模拟中可能出现的值都有绝对的边界，这一点当然正确，但在我们的程序中使用一个 20 乘 60 的网格的确太受限、太武断了。我们有可能编写一个不限制网格大小的 Life 程序，但在这样做之前，我们必须设计几个复杂的数据结构。只有当我们实现这些数据结构以后，我们才能在 9.9 节编写一个通用的不带网格大小限制的 Life 程序。

然而，我们在首次尝试中，对要解决的问题加以限制是完全有道理的，因此，暂时让我们继续研究受限大小的网格上的 Life。尽管如此，准确地说出我们要做的事情仍然非常重要：

> **程序设计技术规则**
>
> 确信你完全地理解了问题。如果必须改变其条件，则确切地解释所做的修改。

2. 程序正确性

由于程序测试只能说明错误的存在，而不能说明没有错误，因此我们需要其他的方法来不容置疑地证明程序的正确性。构造正规有效的证据来证明程序的正确性经常存在困难，但有时候还是能够做到的，就像我们要在后继章节中对所设计的一些复杂算法所做的证明。对 Life 游戏，我们暂且还是满足于用更加非正规的理由来证明程序为何是正确的。

首先，考虑程序的哪些部分需要验证。Life 配置仅由 update 方法修改，并且只有 update 和 neighbor_count 涉及可能出错的任何计算。因此应当集中考虑这两个方法的正确性。

＊＊＊neighbor_count 的正确性

neighbor_count 方法只查看作为参数给出的单元及其邻居。此单元及其邻居的状态只有有限的几种可能性，因此通过为 neighbor_count 构造一个驱动器，对这些可能性进行玻璃盒测试是可行的。这种测试会很快使我们确信 neighbor_count 的正确性。

＊＊＊update 的正确性

对 update 方法，首先我们应当检查 switch 语句中的情况以确信其动作正确地对应 1.2.1 节中的规则。接下来，我们能够注意到每个单元的动作仅仅依赖于这个单元的状态及其邻居计数。因此，就像对 neighbor_count 那样，我们能够构造有限集合的玻璃盒测试数据，验证 update 在每种可能的情况下执行了正确的动作。

3. 用户接口

＊＊＊问题:输入

在运行 Life 程序时,可能会发现用于输入初始配置的低劣方法是主要的不便之处。让人来计算并键入每个活的单元的数字坐标的做法显得很不自然。输入格式应该反映出与打印的配置同样的视觉效果。至少,程序应该允许用户像输入由空白(用于死的单元)和非空白字符(活的单元)组成的行那样输入配置的每一行。

＊＊＊文件输入和输出

Life 配置可以相当复杂,为了更简单地输入,程序应该能从一个文件中读入初始配置。为允许中断程序并在以后再继续运行,程序也应该能够将最后的配置存入文件,以后可以从这个文件中再次读取配置。

＊＊＊编辑

另一项选择是允许用户编辑任意一代的配置。

＊＊＊输出改进

程序的输出也能够得到改进。应该使用直接的光标定位来仅仅修改那些已经改变状态的单元,而不是重新输出每一代的整个配置。可以使用颜色或其他特性来使输出更漂亮更有用。例如,最近变活的单元可以用一种颜色,而那些继续活着的单元可以按其存活的时间长度赋予其他颜色。

＊＊＊帮助窗口

为了使程序更加完备,选择性地显示 Life 游戏的简要描述及其规则也是有用的,或许可以采用一个弹出窗口来显示这些内容。

总的来说,设计的程序要有吸引用户的外观,这一点非常重要。在大型程序设计中,将用户接口看得非常重要,甚至超过程序中所有其他部分的组合。

> **程序设计技术规则**
>
> 最精心地设计用户接口,程序的成功很大程度上是靠它的吸引力和易用性。

4. 模块化和结构

我们已经在原始设计中谈论过这些问题,已做的结论仍然适用。

5. 文档

同样,我们在前面也谈论过文档的问题,不必再在此处重复。

6. 效率

Life 程序的哪一部分花费的运行时间最多?当然不会是在输入阶段,因为输入只做一次。输出通常也相当有效率。大部分计算发生在 update 方法及其调用的 neighbor_count 方法中。

在每一代,update 重新计算每个可能单元的邻居计数。在典型的配置中,可能只有 5 个

单元是活的，并且它们经常位于网格的某个区域中。因此 update 将大量时间花费在艰苦地证实许多死的且没有活的邻居的单元，这些单元实际上因为其邻居单元计数为 0 而在下一代仍是死的。如果有 95%的单元是死的，那么在计算机时间的使用方面效率就相当低下。

* * * 低效率

但这种效率低下的影响重要吗？通常情况下并不重要，因为对用户来说计算是这样快，每一代好像瞬间就出现了。另一方面，如果在一台非常慢的计算机上或者在繁忙的分时系统上运行这个 Life 程序，就可能会发现程序的速度有点令人失望，在打印一代和开始打印下一代之间有明显的停顿。既然这样，就值得试着节省计算机时间。但一般来说，像 Life 这样的程序，即使效率很低，也不必对其进行优化。

程序设计技术规则

除非必要，不要优化代码。

在代码完善和正确之前，不要开始优化代码。

大多数程序将 90%的时间花在 10%的指令上，找出这 10%，集中精力提高它的效率。

在程序开始优化前进行仔细思考的另一个原因是优化经常生成更复杂的代码，这种代码会更难于进行调试和必要的修改。

程序设计技术规则

尽你所能保持算法简单。当犹豫不决时，选择简单的方式。

1.5.3　程序修正和重新开发

当我们继续评价程序，斟酌它是否满足目标和用户需求时，可能会接连发现它在当前设计中的不足和能够使它变得更为有用的新特性，因此程序复查自然导致程序修正和重新开发。

例如，当复查 Life 程序时，我们发现它很好地满足了其中一些标准，但对其他标准却有所不足。其中最为严重的是，由于限制了网格大小，它未能满足游戏说明。它的用户接口也与期望相差甚远。最后，它的计算效率低下，但这可能并不重要。

有了一些想法，我们能简单地改进 Life 程序的用户接口，本节后面有几个项目都在某些方面提议对 Life 进行改进。然而修正程序以移去网格大小的限制会要求使用我们尚未设计的数据结构和算法，因此我们将在 9.9 节再来访问这个 Life 程序。到那时，我们会发现新设计的算法也解决了效率问题。因此新的程序将同时满足通用需求并在计算上更为高效。

程序设计技术规则

有时延缓问题会简化解决方案。

练习 1.5

E1. 有时用户可能希望在小于 20×60 的网格上运行 Life 程序。判断怎样可能将 maxrow 和 maxcol 作为变量，使用户在程序运行时能设置它们。试着尽可能在程序中少做修改。

E2. 对函数 Life::neighbor_count(row, col)进行改进的一种构想是从数组 grid 和 new_grid 中删去栅栏(始终是死单元的额外的行和列)。然后，当单元在边界上时，neighbor_count 检查的单元数会少于 8 个，因为其中有一些已在网格的边界之外。为了这样做，函数需要判断单元(row, col)是否在边界上，但可以将它放在嵌套循环之外来完成，即在循环开始前判断此循环的上下边界。例如，如果 row 与所允许的值同样小，则行循环的下边界为 row；否则，它就是 row-1。按照网格的大小判断，neighbor_count 的原版本和新版本各约执行多少语句。值得做出本练习所提议的修改吗？

程序设计项目 1.5

P1. 修改 Life 的 initialize 函数，使它将空白和 x 构成的一个序列作为适当行中的被占据位置输入，从而设置初始的 Life::gird 配置，而不是要求以数字坐标对来输入被占据位置。

P2. 向函数 initialize 中加入功能部件，使它能够根据用户的选择，从键盘或者从文件读入初始配置。其中文件的第一行是注释，给出了此配置的名称，文件中剩余的每一行对应配置中的一行。每一行中活的位置包含一个“x”，而每个死的位置是一个空白。

P3. 向 Life 程序中加入功能部件，使得它在中断时能将最后配置写入一个文件，文件格式能够被用户编辑，能被读入以重新启动此程序(使用练习 E2 的功能部件)。

P4. 向 Life 程序中加入功能部件，使得在任意一代，用户能够通过插入新的活的单元或删除活的单元来编辑当前配置。

P5. 向 Life 程序中加入功能部件，使得在任意一代，如果用户希望的话，它能显示一个帮助窗口，给出 Life 游戏的规则并解释如何使用此程序。

P6. 向 Life 程序中加入单步模式，使得它能解释从一代到下一代所做的每次变化。

P7. 用直接光标定位(一种依赖于系统的特性)让 Life 的 print 方法更新配置，而不是在每一代完全重写它。

P8. 在 Life 的输出中使用不同的颜色来显示哪些单元在当代发生了变化，哪一些没有变化。

1.6 结论和复习

本章对大量话题做了综述，但主要是概况。其中的一些主题将在后继章节做更深的论述，一些将放到更高级的课程中学习，还有一些通过实践学习则效果最好。

本节扼要重述并展开一些已经学过的原理。

1.6.1　软件工程

软件工程(software engineering)是对有助于大型软件系统的建设和维护的方法的研究和实践。虽然本章所研究的例子按现实标准来看显得太小,但它阐述了软件工程的许多方面的内容。

软件工程始于人们意识到需要很长的过程来获得好软件。它在程序编码之前开始,并持续到程序投入使用后多年的维护。这一持续的过程称为软件的生命期,它能分成如下几个阶段:

* * * 生命期阶段

1. 准确全面地分析问题,确信仔细地规定了所有必要的用户接口。
2. 建立原型并用其试验,直到最后定下所有的说明。
3. 使用数据结构和其他已知功能的算法工具来设计算法。
4. 验证算法是正确的,或者使得它尽量简单而无需证明其正确性。
5. 分析算法以判断它的要求并确信它满足说明。
6. 用合适的程序设计语言对算法进行编码。
7. 用精心选择的数据来测试和评价程序。
8. 必要时细化附加的类和函数并重复前述步骤,直到软件完成并具有全部功能。
9. 优化代码以改进性能,但仅在必要时进行。
10. 维护程序,使它能够满足用户不断变化的需要。

我们在本章和前章的许多篇幅中讨论过其中的大多数主题并加以举例说明,但再次对第一阶段的问题分析和说明给出一些更多的讨论还是适宜的。

1.6.2　问题分析

* * * 说明

问题分析经常是软件生命期中最困难的阶段。这并不是因为实际问题在概念上比计算科学习题更为困难——反过来经常是这种情况——而是因为用户和程序设计员习惯于使用不同的语言。分析员和用户必须就下面这些问题达成共识:

1. 输入输出数据采用什么样格式?数据量有多大?
2. 对处理有特别要求吗?什么样的特别事件需要个别对待?
3. 这些需求会变化吗?如何变化?系统需求的增长有多快?
4. 系统的哪些部分最为重要?哪些部分必须最有效地运行?
5. 如何处理错误的数据?还需要其他什么错误处理?
6. 什么人将使用此软件?对他们有什么培训?什么样的用户接口最好?
7. 软件如何具有可移植性,使得它能够转移到新型设备上?系统必须兼容的其他软件和硬件系统有哪些?
8. 预期的扩展或其他维护是什么?以前对软硬件修改的历史是什么?

1.6.3 需求说明

对大型项目来说，问题分析和试验阶段将最终产生项目需求的正式说明。这个说明成为用户和软件工程师尝试互相理解的主要方式，并且建立了评判最终项目的标准。此说明会包含以下内容：

1. 系统的功能需求：它将要做什么以及用户可用的命令是什么？
2. 系统的假定和限制：系统将使用什么硬件，必须采取什么样的输入格式，输入的最大长度是多少，最大用户数是多少，等等。
3. 维护需求：系统预期的扩展，硬件的变化，用户接口的变化。
4. 文档需求：对哪类用户需要什么种类的解释材料。

需求说明规定软件要做什么，而不是如何做。这些说明对用户和程序设计员都应该是能够理解的。如果精心准备的话，它们能够成为后继设计、编码、测试和维护阶段的基础。

1.6.4 编码

* * * 说明完成

在大型软件项目中，有必要在合适的时间开始编码工作，不宜太早也不宜太迟。多数程序设计员易犯的错误是过早地开始编码。如果在需求说明弄准确之前就进行编码，则在编码时不可避免地会对说明做出无根据的假设，这些假设会使不同的类和函数互不兼容，或者使程序设计任务变得更为困难。

程序设计技术规则

在需求说明准确和完善前不要进行编码。

程序设计技术规则

匆匆行动，慢慢后悔。匆匆编程，常常调试。

* * * 自顶向下的编码

另一方面，可能会将编码延期太长，但实际上这种情况未必会发生。就像我们进行自顶向下的设计，我们应该进行自顶向下的编码。一旦顶层的说明完成且准确，则应该对这些层中的类和函数进行编码并通过加入合适的占位函数来测试它们。如果我们随后发现它的设计中带有缺陷，则可以直接修改它，而不必在未用的低层函数上付出过高的代价。

同样的想法可以稍微表达得更确定些：

程序设计技术规则

重新开始经常比给一个旧程序打补丁更简单。

一条好的经验法则是，如果程序有超过10%的部分必须修改，那么就到了完全重写程序的时候了。对一个大型程序重复打补丁，会使bug的数量趋于一个常数。因为补丁很复杂，

每个补丁可能引入的新错误与它所改正的错误同样多。

一个极好的避免从头开始重写大型程序的办法是在开始时就计划书写两个版本。在程序正在运行时，有可能知道设计的哪些部分会引起困难或需要修改哪些功能部件以满足用户的要求。工程师们多年以来一直知道不可能直接从制图板上建设大型项目。对大型项目，工程师们总是建立原型：即能被研究、测试甚至在某些时候可以用于受限目的的按比例缩小的模型。例如人们在风洞中研究和测试桥梁模型，在装配线上尝试使用新技术之前也会构造试验工场。

＊＊＊软件原型

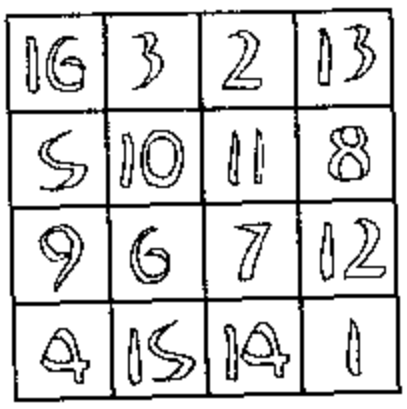
原型法对计算机软件特别有用，因为它减轻了用户和设计者在项目早期的通信，因此减少了误解并有助于使每个人对设计满意。在建立软件原型时，设计者能利用已经写就的输入输出、排序或其他通用需求的程序，但在用构造块装配能完成预期任务的工作模型时要尽可能地少用新的程序设计。即使原型可能不能有效地运转或完成最终系统的每项功能，但是它却为用户和设计者提供了一个极好的实验室来试验可供选择的最终设计思想。

程序设计技术规则

总是计划建立原型并丢弃它，不管是否计划都必须这样做。

程序设计项目 1.6

P1. 幻方是一个方形整数数组，它每行的和、每列的和及两条对角线上的和全部相等。图 1.6 给出了两个幻方。①



和 = 34

17	24	1	8	15
23	5	7	14	16
4	6	13	20	22
10	12	19	21	3
11	18	25	2	9

和 = 65

图 1.5　两个幻方

(1) 编写程序读入一个正方形整数数组，并判断它是否是一个幻方。

(2) 编写程序用下面的方法生成一个幻方，此方法仅当正方形大小是奇数时可用。开始时将 1 放在顶行的中间位置，将后续整数 2、3，…分别写入前个数字所在位置的上行偏右一格。当到达顶行时(比如 1 就放在顶行)，返回到底行继续进行，就好像底行是直接在顶行上面一样。当到达最右列时，继续到最左列，就好像最左列直接在

① 这里所示的左边的幻方出现在 Albrecht Dürer 的铜版画 Melancolia 中，注意包含此铜版画的日期：1514 年。

最右列的右边。当到达的位置已被占据时，就直接在前一个数的下面位置处插入新数。按这种方法构造的 5×5 幻方如图 1.5 中所示。

P2. 一维 Life 发生在一条直线上而非矩形网格内，它的每个单元有 4 个邻居位置：它每侧距离为 1 或 2 的单元。一维 Life 的规则类似于二维 Life，除了规则(1)死的单元如果有 2 或 3 个活的邻居，则它将在下一代变成活的；和(2)活的单元如果有 0、1 或 3 个活的邻居则将死去。(因此有 0、1 或 4 个活的邻居的死的单元仍然是死的；有 2 或 4 个活的邻居的活的单元仍然是活的。)在图 1.6 中显示了样本群落的发展。为一维 Life 设计、编写并测试程序。

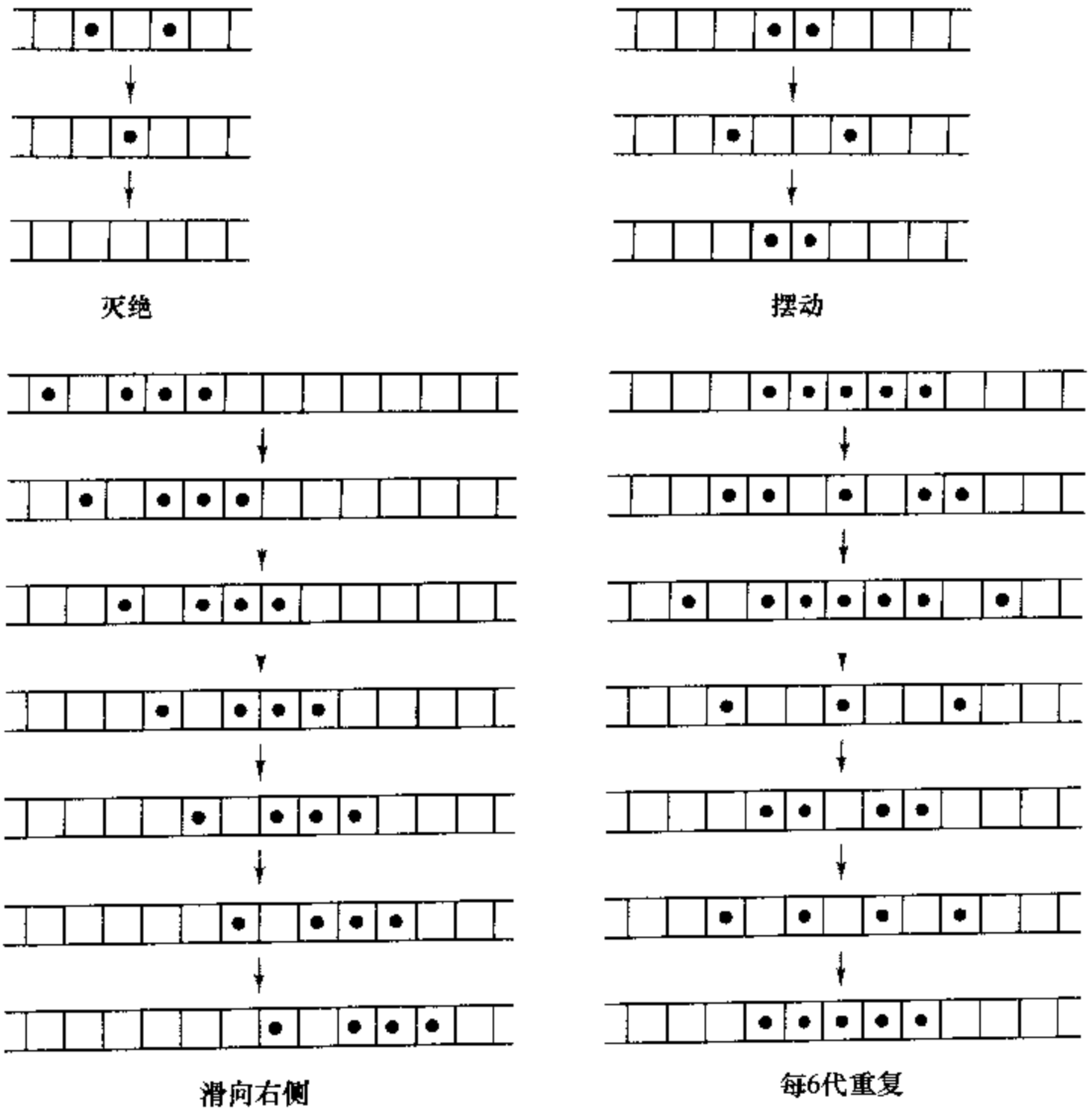


图 1.6 一维 Life 配置

P3. (1) 编写一个程序打印出当年的日历。

(2) 修改此程序，使得它能读入一个年号并打印出那一年的年历。如果此年号是 4 的倍数并且不是世纪年(是 100 的倍数)，除非年号能够被 400 整除，则此年是闰年(二月有 29 天，而不是 28 天)。因此 1900 年不是闰年，而 2000 年是闰年。

(3) 修改此程序，使得它能接受任何日期(日、月、年)并打印出此日期是星期几。

(4) 修改此程序，使得它能读入两个日期并打印出这两个日期之间的天数。

(5) 使用闰年规则，证明日历序列确实每 400 年重复一次。

(6) 一个月的 13 日是星期五的概率是多少(按 400 年的周期)？为什么一个月的 13 日更可能是星期五，而不是一周中的任何其他一天？编写程序计算本世纪有多少个星期五且为 13 日。

启示和易犯的错误

1. 为改进程序，复查它的逻辑。不要基于低劣的算法优化代码。
2. 在程序正确且能够运行之前决不要优化它。
3. 除非绝对必要，不要优化程序。
4. 保持函数简短，任何函数很少超过一页长。
5. 开始编码前确信算法是正确的。
6. 验证算法中复杂的部分。
7. 保持简单逻辑。
8. 确信在决定如何求解问题之前理解它。
9. 确信在开始程序设计前理解算法的方法。
10. 如果困难，则将问题划分成多块并分别考虑每一部分。
11. 在描述问题时产生的名词为问题的解答提议了有用的类，动词则提议了有用的函数。
12. 在编写每个函数时包含细致的文档(如出现在 1.3.2 节中的)。
13. 确信为每个函数写下准确的前置条件和后置条件。
14. 在函数开始时包含错误检查以证明前置条件确实成立。
15. 在每次使用函数前，自问为何知道其前置条件会被满足？
16. 使用占位程序和驱动器、黑盒和玻璃盒测试来简化调试。
17. 使用大量的支架来帮助定位错误。
18. 在带数组的程序设计中，谨慎对待比 1 小的下标值，总是使用极值测试来检查使用数组的程序。
19. 在编写程序时保持良好的格式——它会使调试更加简单。
20. 保持文档与代码一致，而且在阅读程序时确信是在调试代码而不仅仅是注释。
21. 向其他人解释程序，这样做会有助于自己更好地理解程序。

复习题

本书大多数章节在结束时都设计了帮助复习本章主要思想的问题集。对这些问题，可以由书中的讨论直接回答；如果对答案不确定的话，请参考合适的章节。

1.3

1. 何时适合使用单字母的变量名？
2. 指出 4 类应包含在程序文档中的信息。
3. 外部和内部文档的区别是什么？
4. 什么是前置条件和后置条件？

5. 指出三类参数，C++中如何处理它们？
6. 为什么应避免函数的副作用？

1.4

7. 什么是程序占位？
8. 占位程序和驱动器之间的区别是什么，分别在何时使用它们？
9. 什么是结构化预排？
10. 什么是程序中的支架？何时使用支架？
11. 指出一种实施防御式程序设计的方法。
12. 给出两种测试程序的方法，并讨论应在何时使用哪一种？
13. 如果不能立即描绘出解决问题所需要的所有细节，那么应该如何处理此问题？
14. 子程序的前置条件和后置条件是什么？
15. 何时应该在函数间分配任务？

1.6

16. 编码应该延缓多长时间？
17. 什么是程序维护？
18. 什么是原型？
19. 指出软件生命期至少 6 个阶段并说明每个阶段是什么状态。
20. 定义软件工程。
21. 什么是程序的需求说明？

进阶参考书目

C++

程序设计语言C++由 Bjarne Stroustrup 发明，他最初在 1984 年发表了C++的描述。标准参考手册是 B. Stroustrup，*The C++ Programming Language*，third edition，Addison Wesley，Reading，Mass.，1997.

有许多好的课本对C++提供了更通俗的描述，这类书太多，再此就不一一列出了，这些课本也提供了许多例子和应用。

对已经知道这门语言的程序设计者，推荐一本关于如何有效地使用C++的有趣的书：Scott Meyers，*Effective C++*，second edition，Addison-Wesley，Reading，Mass.，1997.

程序设计原理

有两本书包含关于程序设计风格和正确性方面许多有用的提示，也包含正确的和不良的实践的例子，分别是：Brian Kernighan and P. J. Plauger，*The Elements of Programming Style*，second edition McGraw-Hill，New York，1978，168 pages. Dennie Van Tassel，*Program Style，Design，Efficiency，Debugging，and Testing*，second edition，Prentice Hall，Englewood Cliffs，N. J.，1978，323 pages.

Edsger W. Dijkstra 是结构化程序设计的先驱，他强调在程序设计和程序编写时采用精心组织的自顶向下的方法。在 1968 年 3 月，他因为在 *Communications of the ACM*（vol. 11，第 147～148 页）上发表题为“Go To Statement Considered Harmful”（Go To 语句是有害的）的信件而引起一些震惊。自那以后，Dijkstra 已发表了数篇论文和书籍，它们是程序设计方法中最有启发性的。特别重要的一本书是：Edsger W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N. J., 1976, 217 pages.

全面介绍面向对象程序设计的一本书是：Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, Calif., 1994.

Life 游戏

杰出的英国数学家 J. H. Conway 在有限单群、逻辑和组合数学理论的多个学科做出了许多独创的贡献。他从先前的关于元胞自动机（cellular automata）的技术研究开始设计 Life 游戏，然后设计繁殖规则使得配置难以无边界地生长，并使得许多配置能经历有趣的演进。然而，Conway 自己并没有发表他的观察结果，而是将它告知了 Martin Gardner。当这个游戏在下面的文章中被讨论后，它快速流行起来。

Martin Gardner, “Mathematical Games” (regular column), *Scientific American* 223, No. 4 (October 1970), 120-123; 224, no. 2 (February 1971), 112-117.

1.2 节和 1.4 节结尾处的例子来自这些专栏，这些专栏和更多的相关的成果被重新再版于：Martin Gardner, *Wheels, Life and Other Mathematical Amusements*, W. H. Freeman, New York and San Francisco, 1983, pp. 214-257.

这本书也包含一篇关于 Life 的文章的参考书目。甚至当时有几年还出了一份名为 *Lifeline* 的新闻季刊，它跟踪实际爱好者在 Life 和相关话题方面的最新的研究进展。

软件工程

下面这本书对结构化程序设计的许多方面进行了全面的讨论：Edward Yourdon, *Techniques of Program Structure and Design*, Prentice-Hall, Engle-wood Cliffs, N. J., 1975, 364 pages.

下面这本书提供了关于大型软件系统构造中产生的许多问题的细致讨论（这本书也是一本有趣的读物）：Frederice P. Brooks, JR., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Mass., 1975, 195 pages.

关于软件工程的一本好课本是：Ian Sommerville. *Software Engineering*, Addison-Wesley, Wokingham, England, 1985, 334 pages.

* * * 算法验证

两本关于证明程序并使用断言和不变式来设计算法的书是：David Gries, *The Science of Programming*, Springer-Verlag, New York, 1981, 366 pages. Suad Alagić and Michael A. Arbib, *The Design of Well-structured and Correct Programs*, Springer-Verlag, New York, 1978, 292 pages.

保持程序简单以能够证明它们的正确性并非易事，但这一点却很重要。C. A. R. Hoare（他发明了我们将在第 8 章中学习的快速排序算法）写道："有两种构造软件设计的方法：一种方法是使它非常简单而明显没有缺陷，另一种方法是使它如此复杂而没有明显的缺陷，第一种方法远远比第二种方法困难。"这句引语来自他 1980 年的图灵奖讲演："The emperor's old clothes," *Communications of the ACM*, 24(1981), 75-83.

* * * 问题求解

两本关于问题求解方法的书是：George Pólya, *How to Solve It*, second edition, Doubleday, Garden City, N. Y., 1957, 253 pages. Wayne A. Wickelgren, *How to Solve Problems*, W. H. Freeman, San Francisco, 1974, 262 pages.

一维 Life 程序设计项目取自 Jonathan K. Miller, "One-dimensional Life," *Byte* 3 (December, 1978), 68-74.

第2章　栈

本章引入栈的学习，栈是所有数据结构中最简单但却是最重要的一种。本章用一个访问标准库栈实现的程序来详细地说明如何将栈用于反转数据，然后设计了栈数据结构的顺序实现并且将它用于实现逆波兰计算器和括号匹配程序。本章以对抽象数据类型和数据结构的普遍原理的讨论结束。

2.1　栈说明

2.1.1　表和数组

在介绍了循环和数组后，每个初级程序设计课程就会给出如下的程序设计练习：

读取一个整数 n，它最大为 25，然后读取 n 个数的表，并且以逆序打印这个表。

这个简单的练习可能给某些学生带来困难。大多数学生认为需要使用一个数组，但有些学生试图建立一个有 n 个元素的数组，并对由于试图使用变量而不是常数来声明定义数组的大小所引起的错误信息感到迷惑。其他的学生会说，“如果我知道有 25 个数，那么我就可以解决这个问题，但如果少于这个数字，我就不知道如何处理了。”或者说“在我写程序之前告诉我 n 有多大，我就可以做了。”

＊＊＊表和数组

这些学生的困难不是因为笨，而是因为逻辑思维。初级课程有时没有足够地区分两个非常不同的概念，其中第一个概念是 n 个数的表的概念，一个表的大小是可变的。也就是说，对一个表来说，表中的元素可以插入也可以删除。因此，如果 $n=3$，那么这个表仅仅包含了 3 个元素；如果 $n=19$，那么表中包含 19 个元素。第二个概念是称为数组或向量的程序设计部件，数组或向量包含有常数个位置。也就是说，在程序编译时它的大小是固定的。表是动态的数据结构，因为它的大小能够改变，而数组是静态的数据结构，因为它有固定的大小。

＊＊＊实现

当然，表的概念和数组的概念是相关的，这是因为大小可变的表在计算机中可以作为占据固定长度的数组的一部分实现，而数组中可能留有一部分未被使用。然而，我们随后将发现，有几种不同的方法可以实现表，因此，在选择和描述数据结构时，我们不应当混淆实现上的决策和更基本的决策。

2.1.2　栈

栈是一种形式的表，它对内含反转的应用特别有用，如 2.1.1 节的问题。在栈这种数据结构中，所有的元素插入和删除都在表的一端进行，这一端称为栈顶(top)。一个有用的类

比(见图 2.1)是将栈看作是繁忙的自助餐厅的柜台上堆叠起来的碟子或盘子,在整个用餐时间内,顾客取走堆顶上的碟子,雇员将返回的碟子放在堆顶上。最近放在堆上的碟子就是最先被拿走的碟子,最底下的碟子是第一个放上去,也是最后一个被使用的。

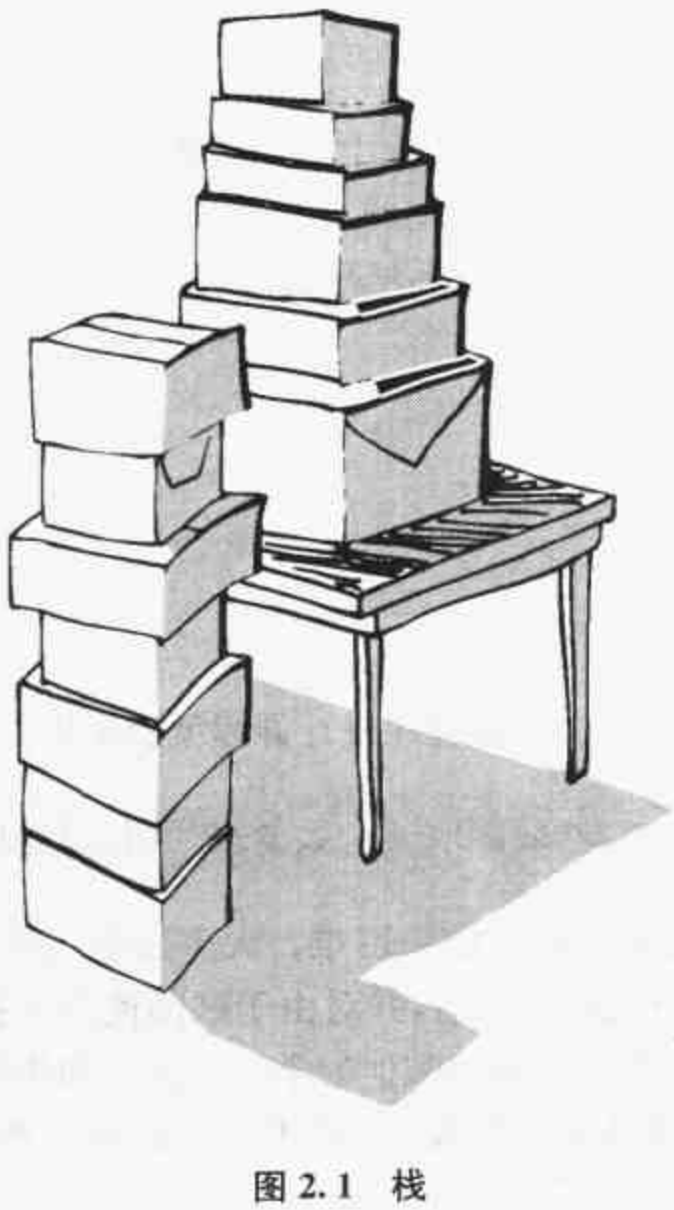


图 2.1 栈

* * * push 和 pop

有时这幅图会被描绘成放在一个带弹簧的装置上的盘子或碟子,因此堆顶总是停留在近乎同一高度,但这样的想像是差劲的而且应当予以避免。如果我们以这种方式实现计算机上的栈,这就意味着无论何时插入或删除栈中的一项时都会移动栈中的每一项。把栈想像成是放在固定的柜台上更好,使得对栈有插入或删除时只移动栈顶的项。然而,带有弹簧的栈想像提供了一对有趣的术语,它们已牢牢地植根于计算机的行话,我们也将用它们命名基本的栈操作。当向栈中增加一项时,称之为入栈(push),当从栈中删除一项时,称之为出栈(pop),见图 2.2。注意:最后压入到栈中的项总是最先从栈中弹出的项。这种特征称为后进先出(last in, first out),简称为 LIFO。

2.1.3 第一个例子:反转表

作为使用栈的一个简单例子,我们编写一个程序解决 2.1.1 节中的问题。程序读入一个整数 n 和 n 个浮点数,然后以相反的顺序写出。为完成这个任务,可以在读取每个数据时将它入栈;当输入完成时,将数据出栈,即可以实现反序。

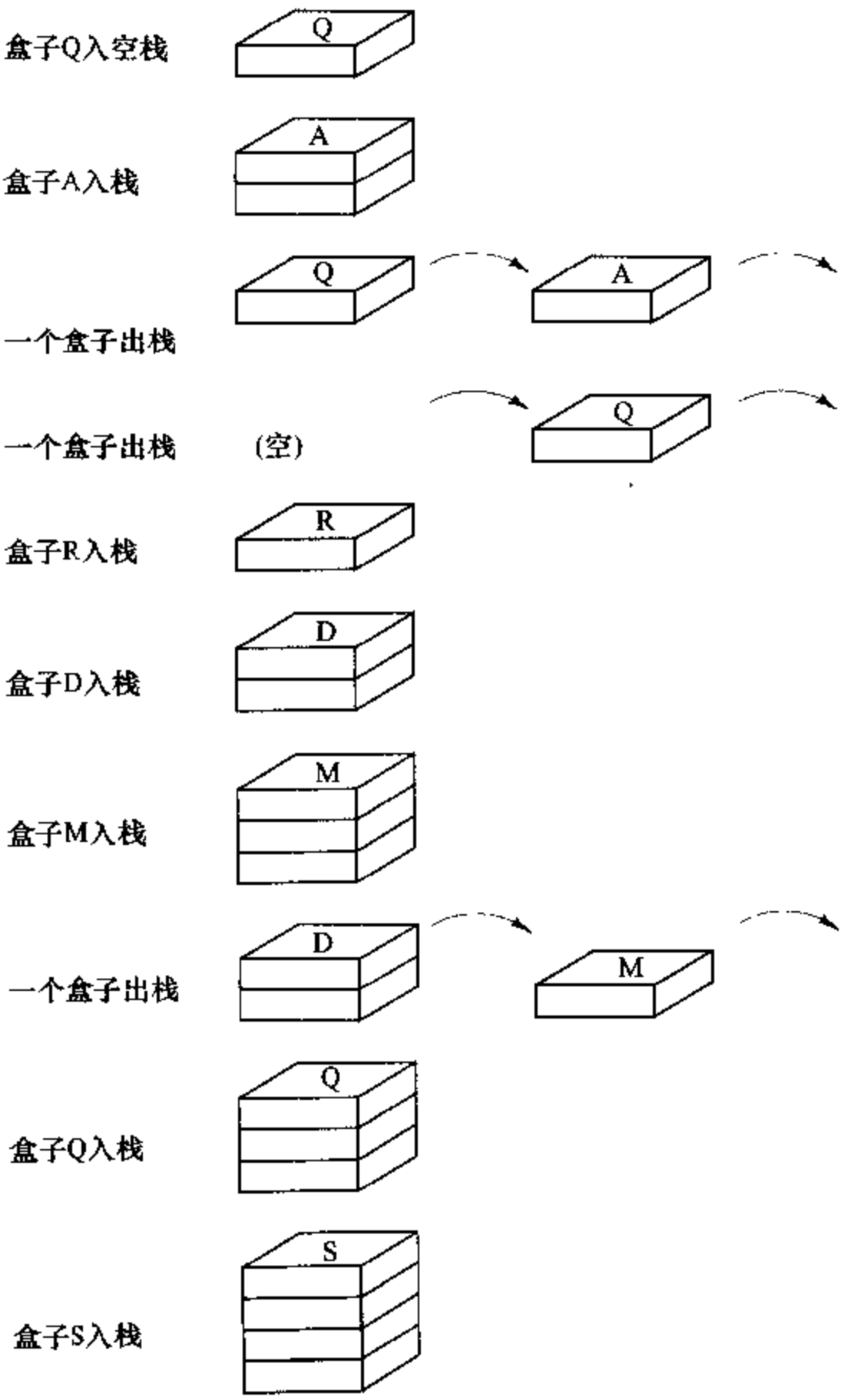


图 2.2 入栈和出栈

* * * 标准模板库

在程序中,我们将依赖于C++的标准模板库(通常称为 STL)提供一个类来实现栈。[①] STL 是C++标准库的一部分,这个标准库包含了各种有用的信息、函数和类。STL 是标准库的一部分,它为许多普通的数据结构提供了方便的实现,包括这本书中我们将学习的几乎所有的数据结构。

可以用命令 # include< stack>(或 ANSI 之前的老一点的版本,使用 # include< stack.

① 如果 STL 栈实现不可用,可以使用在下一节实现的栈类。

h>)将 STL 栈包含进我们的程序中。一旦包含了库,就可以定义并初始化置空的栈对象,并且运用 push,pop,top 和 empty 方法。稍后将详细讨论这些方法和 STL 本身,下面的程序直接使用了它们。

```
# include < stack>

int main()
/*Pre: The user supplies an integer n and n decimal numbers.
  Post: The numbers are printed in reverse order.
  Uses: The STL class stack and its methods */
{
  int n;
  double item;
  stack<double >numbers;   //  declares and initializes a stack of numbers
  cout<<"Type in an integer n followed by n decimal numbers."<<endl
      <<"The numbers will be printed in reverse order."<<endl;
  cin>>n;
  for (int i=0; i<n; i++){
    cin>>item;
    numbers.push(item);
  }
  cout<<endl<<endl;
  while (!numbers.empty()){
    cout<<numbers.top()<<" ";
    numbers.pop();
  }
  cout<<endl;
}
```

* * * 初始化

在这个反转数的程序中,不仅使用了称作 numbers 的栈的方法 push(),top()和 pop(),而且重要的一点是暗示了栈 numbers 是一个空栈。也就是说,当称为 numbers 的栈被创建时,自动地初始化为空。比如对标准库类,只要创建一个类,我们就应谨慎,保证类自动地初始化,而变量和数组的初始化必须是显式的。

* * * 大写

要提及的是,像 int,float 等原子类,在C++库的栈类中有以小写字母开头的标识符。然而就像在 1.3 节中约定的,我们要创建的类的标识符将以大写字母开头。

* * * 模板

STL 栈实现的一个重要特征是用户可以说明特定栈中存放的数据类型,如在反转程序中,我们用声明 stack<**double**>numbers 创建了一个元素类型为 double 的栈,而如果需要一个整型的栈,则可以声明为 stack<**int**>numbers。标准库用称为模板的C++的构造去实现这种灵活性,一旦熟悉了数据结构更基本的实现,我们将实践这种构造并使用自己的模板,这将从第 6 章开始。

2.1.4 信息隐藏

我们已经可以写出用来反转一行输入的程序,而不用考虑栈是如何真正实现的。以这

种方式，我们就有了一个信息隐藏的例子：处理栈的方法在C++标准库中实现，但我们可以使用这些方法，而不需知道栈在内存中是如何存储的或者栈是如何真正操作的等细节内容。

*** 内置结构

事实上，我们在先前编写的程序中未经考虑就实行了信息隐藏。无论何时使用数组或结构编写程序，这些结构的操作就已满足我们的需要，因此不必去考虑C++编译器在计算机内存中如何以比特或字节去表示它们，也不必考虑它如何遵循机器语言的步骤去查找一个下标或者选择一个成员。

*** 可选的实现

对数组实行信息隐藏和对栈实行信息隐藏之间的一个重要区别在于C++只提供了数组的一种内置实现，而 STL 有几种栈的实现。虽然使用栈的客户程序代码不应该依赖于栈实现的特定选择，但是最终程序的性能可能在很大程度上依赖于栈的实现。为了对给定应用中将使用哪种栈实现做出有根据的决策，我们需要评价不同实现的不同特征和性能。在随后的章节里，我们将看到栈(同我们要研究的几乎所有的数据类型一样)在计算机内存里有不同的方法可以表示数据，而且有不同的方法进行操作。在一些应用里，一种实现可能比较好，而在另外的应用里，另一种实现可能会证明更优。

*** 实现的改变

即使在一个单一的大型程序里，刚开始时我们可能决定采用某种方法去表示栈，而当有了程序设计经验后，我们可能决定采用另一种更好的方式。如果每次在使用栈的时候都需要设计操作栈的指令，那么这些指令的每一次出现都将需要修改。而如果对栈操作使用独立的函数实行信息隐藏，则仅仅需要改变声明。

*** 程序清晰度

*** 自顶向下的设计

在使用栈的程序里，信息隐藏显示出的另一个好处是，push 和 pop 的适时出现会立即提醒阅读程序的人程序所做的事情，而指令本身可能更晦涩。我们会发现，将数据结构的使用从实现中分离将有助于我们同时改进数据结构和程序的自顶向下的设计。

2.1.5　标准模板库

*** 数据结构库

标准的C++库在 ANSI C++的实现中是存在的，这个库提供了各种各样与系统有关的信息，如能以浮点型存储的最大指数、输入输出工具和那些依赖于系统进行最佳实现的其他函数。另外，标准库提供了大量的数据结构以及用于编写程序的方法。实际上，标准库包含了本书中考虑的几乎所有的数据结构的实现，包括栈、队列、双端队列、表、字符串和集合等。

为了能正确、高效地使用这些库的实现，有必要学习标准库中数据结构的原理和可选的实现，因此我们仅仅简单地介绍标准库，然后返回到我们的主要目标、即数据结构本身。然而，在某种意义上，这本书的大部分可以看作是介绍C++的 STL，因为我们的目标是学习数据结构的基本原理和方法，这是领悟使用 STL 的必不可少的知识。

* * * 模板参数

* * * 可供选择的实现

就像我们已经指出的，STL 栈实现是一个类模板，因此程序设计员能够通过在< > 之间说明模板参数，正确地选择将哪些项放在栈中。实际上，程序设计员也能利用第二个模板参数去控制将要使用哪一种栈实现。第二个参数有一个默认值，因此自己不能确定要使用哪种实现的程序设计员可以利用默认的实现来获得栈的构造，实际上它是来自于一个双端队列——我们将在第 3 章介绍这种数据结构。程序设计员可以选择基于向量或基于表的实现以代替使用栈。为了在这些实现间作出明智的选择，程序设计员需要理解它们的相对优点，并且这种理解只能来自于本书中对各种数据结构进行的那些普遍研究。

* * * 算法性能

然而，不管选择哪种实现，STL 一定会保证栈的方法高效地执行，在常量时间内运行，并独立于栈的规模。我们将在第 7 章对各种算法使用的时间进行系统的研究，并且将在其后继章节里继续进行这种研究。就像所发生的，标准栈方法常量时间内的操作只能在平均意义上，已知的平摊性能上得到保证，我们将在 10.5 节中研究平摊分析。

STL 提供了许多其他的标准数据结构的实现，在学习本书的过程中，我们会注意到它们与所讨论的主题相关的实现。一般来说，这些库实现是高效的、方便的，并且在设计中采用了许多默认的选择，使得程序设计员能自如地使用它们。

练习 2.1

E1. 如图 2.2 画出一系列栈帧，每一个小题都以空的栈 s 开始，并要求栈帧序列能够反映出下列每段代码的执行过程。假设声明如下：

```
# include<stack>
stack<char>s;
char x, y, z;
```

```
(1) s.push('a');
    s.push('b');
    s.push('c');
    s.pop();
    s.pop();
    s.pop();
```

```
(2) s.push('a');
    s.push('b');
    s.push('c');
    x= s.top();
    s.pop();
    y= s.top();
    s.pop();
    s.push(x);
    s.push(y);
    s.pop();
```

```
(3) s.push('a');
    s.push('b');
    s.push('c');
    while(! s.empty())
    s.pop();
```

```
(4) s.push('a');
    s.push('b');
    while(! s.empty()){
    x= s.top();
    s.pop();
    }
    s.push('c');
    s.pop();
    s.push('a');
    s.pop();
    s.push('b');
    s.pop();
```

E2. 编写一个程序，利用栈读入一行文本，然后以反序输出行中的字符。

E3. 编写一个程序，读入以升序排列的一系列整数，然后以降序打印出这些整数。一旦读到的整数不大于前面的数，则输入终止，然后将这些整数以降序打印输出。

E4. 栈可以看作如图 2.3 的铁道交换网。编号为 1,2,…,n 的车厢停在左车道上，当车厢停在右车道上时，要对它们进行重新排列(改序排列)。可以将支线(栈)上的车厢留在那儿或者发送到右车道上，但不能将它再送回到进来时的车道。例如，如果 $n=3$，并且我们将车厢 1,2,3 停在左车道上，则车厢 3 最先到达支线。然后我们可以将车厢 2 发送到支线，然后放在去右车道的路上，再将车厢 3 发送上路，然后是车厢 1，因此获得的新序列为 1,3,2。

(1) 如果 $n=3$，找到所有可能获得的排列。

(2) 如果 $n=4$，找到所有可能获得的排列。

(3) 挑战：对广义的 n，找出使用这个栈可能得到多少种排列。

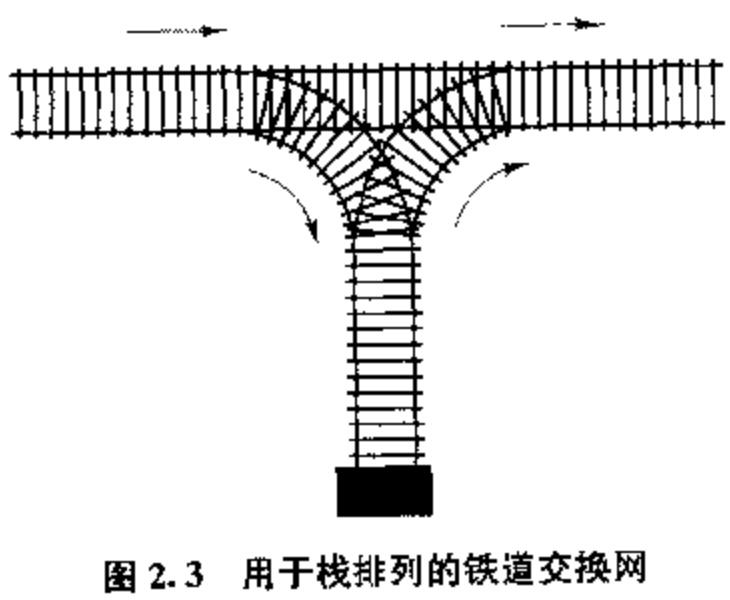


图 2.3　用于栈排列的铁道交换网

2.2　栈的实现

＊＊＊顺序实现

我们现在开始讨论C++中栈实现的构造问题，并提出顺序(contiguous)Stack 的实现，这意味着元素在数组中是一个接一个地存储。在第 4 章，我们将研究在动态内存中使用指针的链式实现。

＊＊＊类

在我们构造的实现及所有其他的实现中，总是谨慎地利用类来实现数据结构。因此，我们现在设计一个 Stack 类，它的数据成员表示栈的元素。但在我们实现任何类之前，应首先确定它的方法的说明。

2.2.1　栈方法的说明

＊＊＊栈方法

Stack 类的方法必须包含名为 empty(),top(),push()和 pop()的基本操作，另外，仅有一个其他的操作是必需的：即设置空栈的初始化操作。若没有这样的初始化操作，客户程序代码将不得不处理由随机数据或可能非法的数据组成的栈，这些非法的数据在处理之前存储在栈所占用的存储区域内。

1. 构造函数

C++语言允许对任何类定义特殊的初始化方法，这些方法称为类的构造函数。每一个构造函数是一个与相应的类具有同样名称的函数。构造函数没有返回类型，只要声明了这个类的对象，它就会自动应用构造函数。例如，栈的标准库实现包括构造函数，它将每个新近创建的栈初始化为空，在我们先前的反转输入的程序中，采用这样的初始化是极其关键的。我们自然应该为自己开发的类创建一个类似的 Stack 构造函数，因此，只要客户程序定义了一个 Stack 对象，那个对象就会被自动地初始化为空。下面是 Stack 构造函数的说明。

* * * 初始化

```
Stack::Stack();
前置条件：无。
后置条件：Stack 存在且被初始化为空。
```

2. 元素类型与泛型

* * * 元素类型

栈的基本方法的定义依赖于我们想存储在栈里的元素类型。为了尽可能地保持通用性，我们用 Stack_entry 作为 Stack 栈中元素的类型。对于某一个应用，Stack_entry 可能是 int，对另一种应用则可能是 char。客户程序可以用如下的定义选择一种合适的元素类型：

```
typedef char Stack_entry;
```

通过保持类型 Stack_entry 的通用性，我们能将同样的栈实现用于许多不同的应用中。

* * * 泛型

* * * 模板

对不同的元素类型使用同样的基本数据结构和操作的能力称为泛型(generic)。我们使用 typedef 语句选择栈 Stack 里的元素类型即是实现C++的泛型数据结构的一种简单方法。复杂的应用可能需要在单个程序中有几个不同元素类型的栈，而用于标准库类的栈更适合使用更复杂的模板去处理。当我们已经获得关于简单数据结构的一些经验以后，我们将在第 6 章开始选择使用模板。

3. 错误处理

* * * 错误代码

在决定基本的 Stack 方法的参数和返回类型时，必须认识到一个方法可能会被客户程序非法地使用，如客户程序可能试图弹出一个空栈。因此我们的方法应该能够用诊断性的错误代码(error codes)指明这样的问题。在本书中，我们使用单一的称为 Error_code 的枚举类型来报告所有来自于程序和函数的错误。

枚举类型 Error_code 是我们的实用程序包的一部分，我们将它放在附录 C 中进行描述。在实现 Stack 方法时，将使用 3 个 Error_code 值，即

success(成功)，overflow(上溢)，underflow(下溢)

*** 栈错误代码

如果一个方法能正常地完成工作，那么它将返回 success 作为其 Error_code，否则，将返回一个代码去指示发生何种错误。因此，试图弹出空的 Stack 的客户程序将获得一个表示 underflow 的 Error_code。然而，pop 方法的任何其他的应用是合法的，而且将产生一个表示 success 的 Error_code。

*** 错误处理

这是我们进行错误处理的第一个例子，错误处理是一种重要的安全措施，无论何时都应当嵌入在数据结构中。可以选择几种不同的方法去处理数据结构的方法中检测到的错误条件，如直接处理错误，或者打印出错误消息，或者停止程序的运行。作为选择，考虑到方法总是在客户程序里被调用，因此我们可以将错误代码返回给客户程序以让它决定如何处理此错误。我们认为，当检测到错误时，客户程序是判断该如何处理的最好位置，因此我们采用了第二种处理错误的方法。在某些情况下，客户程序代码可以通过立即停止操作来回应错误代码，但在其他情况中可能有必要忽略错误条件。

程序设计技术规则

客户程序在使用了一个类方法后，应决定是否检查发生的错误状态。

类应该设计成让客户程序来决定如何响应错误。

*** 异常处理

C++确实提供了一种更先进的称为异常处理(exception handling)的技术，当检测到一个错误时即抛出一个异常，客户程序然后能捕获到这个异常。这样的异常处理就符合让客户程序决定如何响应数据结构中检测到的错误这一思想。栈和其他类的标准库实现使用异常处理去处理错误条件。然而，考虑到简单性，我们将把课本中所有的实现改为返回错误代码。

4. 方法说明

*** Stack 方法

下面是对 Stack 基本方法的说明。

```
Error_code Stack::pop();
```

前置条件：无。

后置条件：如果 Stack 非空，则删除 Stack 的栈顶元素，如果 Stack 为空，则返回下溢的 Error_code 并且 Stack 未被修改。

```
Error_code Stack::push(const Stack_entry &item);
```

前置条件：无。

后置条件：如果 Stack 不满，则将元素加到 Stack 的栈顶，如果 Stack 是满的，则返回上溢的 Error_code 且栈未被修改。

传递给 push 的参数 item 是一个输入参数，通过将它声明成 const 引用加以指示。相反，下一个方法 top 的参数是一个输出参数，我们通过引用来实现它。

```
Error_code Stack::top(Stack_entry &item) const;
```
前置条件：无。
后置条件：非空的 Stack 的栈顶元素被复制到 item。如果 Stack 为空则返回 fail。

追加在这个方法的声明后面的修饰符 const 表明相应的 Stack 对象不能被方法改变或在方法期间被改变。如同把输入参数指定为常量的重要性，如同信息对于阅读者和编译器的重要性，我们用这个修饰符来指示常量方法也很重要。最后一个 Stack 方法 empty 也被声明为常量方法。

```
bool Stack::empty() const;
```
前置条件：无。
后置条件：如果 Stack 为空，则返回结果 true，否则返回 false。

2.2.2 类说明

* * *栈类型

对于顺序 Stack 实现，应创建一个数组来存放栈里的元素，并用一个计数器来表示栈里的元素个数。我们将这些数据成员和方法收集在一起形成下面的 Stack 的定义，Stack 所包含的元素类型为 Stack_entry，这个定义构成了文件 stack. h。

```
const int maxstack=10;      //  small value for testing

class Stack {
public:
  Stack();
  bool empty() const;
  Error_code pop();
  Error_code top(Stack_entry &item) const;
  Error_code push(const Stack_entry &item);
private:
  int count;
  Stack_entry entry[maxstack];
};
```

正如 1.2.4 节中所做的解释，我们应该将这个类定义放在扩展名为.h 的头文件中，本例中是文件 stack.h。相应的代码文件将称为 stack.c，里面有我们下一步要开发的方法的实现。代码文件然后可以被独立地编译并按需连接到客户程序。

2.2.3 入栈、出栈和其他方法

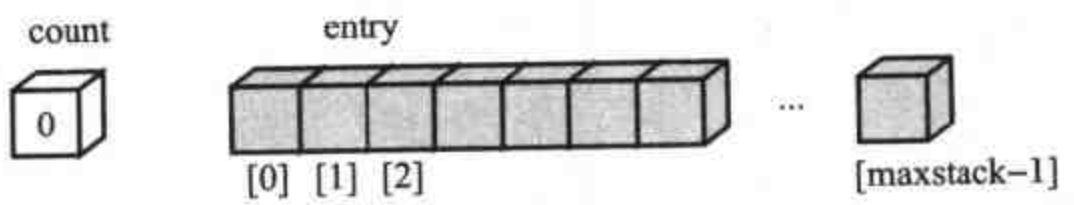
栈的方法如下实现。我们应当小心极端情况：从空栈中弹出一个元素或向已满的栈中

压入一个元素，必须认识到这些情况并用返回的错误代码加以报告。

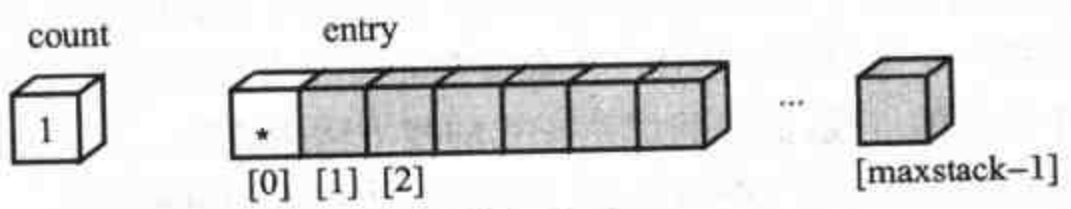
```
Error_code Stack::push(const Stack_entry &item)
/*Pre: None.
  Post: If the Stack is not full, item is added to the top of the Stack. If the Stack is
        full, an Error_code of overflow is returned and the Stack is left unchanged.
        */
{
  Error_code outcome=success;
  if (count>=maxstack)
     outcome=overflow;
  else
    entry[count++]=item;
  return outcome;
}

Error_code Stack::pop()
/*Pre: None.
  Post: If the Stack is not empty, the top of the Stack is removed. If the Stack is
        empty, an Error_code of underflow is returned. */
{
  Error_code outcome=success;
  if (count==0)
     outcome=underflow;
  else --count;
  return outcome;
}
```

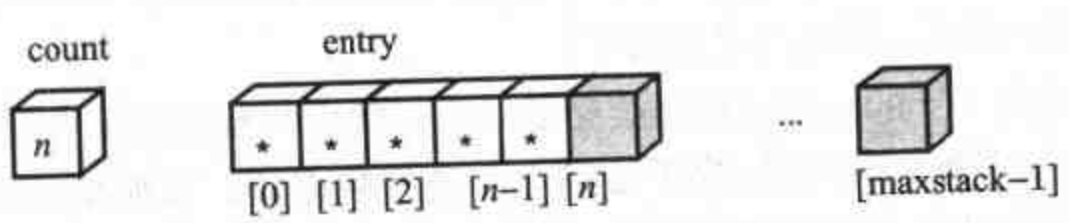
注意：数据成员 count 表示栈中的元素个数，因此栈顶元素为 entry[count-1]，如图 2.4 所示。



(a) 栈为空



(b) 压入首个元素



(c) 栈中的n个元素

图 2.4　顺序栈中的数据表示

```
Error_code Stack::top(Stack_entry &item)const
/*Pre: None.
  Post: If the Stack is not empty, the top of the Stack is returned in item. If the Stack
        is empty an Error_code of underflow is returned. */
{
  Error_code outcome=success;
  if (count==0)
     outcome=underflow;
  else
    item=entry[count-1];
  return outcome;
}
bool Stack::empty()const
/*Pre: None.
  Post: If the Stack is empty, true is returned. Otherwise false is returned. */
{
  bool outcome=true;
  if (count>0)outcome=false;
  return outcome;
}
```

*** 构造函数

Stack 的另一个方法是构造函数,构造函数的目的是将新的栈对象初始化为空。

```
Stack::Stack()
/*Pre: None.
  Post: The stack is initialized to be empty. */
{
  count=0;
}
```

2.2.4 封装

*** 数据完整性

注意到我们的栈实现方法迫使客户程序使用信息隐藏。对数据私有可见性的声明使得客户程序除了用正式的方法 push(),pop()和 top()以外,不能够以其他方法获得栈中存储的数据。数据私有的一个重要结果是栈决不会包含非法的或是被破坏的数据。每个栈对象将被初始化为一个合法的空栈,而且只能由正式的栈方法来修改。只要正确地实现了我们的方法,就能保证已经正确初始化的对象必定不会受到任何数据的破坏。

*** 封装

我们给予 Stack 对象的这种保护可以概括地说成它们被封装。一般来说,如果数据只由受控的函数集进行访问,则说它受到封装。

稍做努力去封装C++的类数据成员会受益匪浅。使用封装类的第一个益处出现在说明和设计方法时:对一个被封装的类,决不必担心非法的数据值。而如果没有封装,那么在数据结构上进行的操作几乎总是依赖于这个前提,即数据成员已经被正确地初始化并且尚未受到破坏,我们能够而且应当避免这种前提条件。对于封装的类 Stack,所有方法的前置条件都是"无",这就意味着客户程序不需要在使用公共栈方法之前检查任何特殊条件,比如未

初始化的栈。既然把数据结构看成是一次写成并且用在许多不同的应用里的服务，那么让客户程序尽可能地免除额外工作的思想是极其合适的。

程序设计技术规则

数据结构的公有方法应当不带前提地予以实现，而数据成员应当保持私有。

我们应当从所有封装的C++类的公有方法说明中省略其前置条件部分。

客户程序不能使用数据结构的私有成员函数，因此不再有强有力的案例来说明需要不带前置条件地编写这些函数。我们应该通过将术语“方法(method)”保留给公有成员使用，以此强调公有和私有成员函数之间的区别。

练习 2.2

E1. 假设下面的定义文件是一个扩展的栈数据结构的顺序实现。

```
class Extended_stack {
public:
  Extended_stack();
  Error_code pop();
  Error_code push(Stack_entry item);
  Error_code top(Stack_entry &item) const;
  bool empty() const;
  void clear();          //  Reset the stack to be empty.
  bool full() const;     //  If the stack is full, return true; else returen false;
  int size()const;       //  Return the number of entries in the stack.
private:
  int top;
  Stack_entry entry[maxstack];
}
```

为下列方法编写代码(在代码中使用私有数据成员)。

(1) clear　　(2) full　　(3) size

E2. 采用栈的方法，为下面的说明编写一个函数 copy_satck：

```
Error_code copy_stack(stack &dest, Stack &source);
```

前置条件：无。

后置条件：Stack dest 已经成为 Stack source 的一个完全副本，而 source 未被改变。

如果检测到错误，则返回一个合适的代码，否则返回 success。

写出 3 个版本的函数：

(1) 仅使用一个简单的赋值语句：dest=source；

(2) 使用 Stack 方法和一个临时的 Stack，从 Stack source 中抽取元素并将每个元素加入到 Stack dest 中，并恢复 Stack source。

(3) 为 Stack 类编写一个友元(friend)[①]函数,使用 Stack 的私有数据成员并且写一个循环将元素从 source 复制到 dest。

这些函数中的哪一个最容易编写?如果栈接近满时,哪一个运行最快?如果栈接近空时,哪一个运行最快?如果实现可被改变,哪一个是最好的方法?哪一个函数可以将 source 作为一个常量引用传递?

E3. 为下列函数编写代码(必须使用栈方法,但不用假设栈及其方法是如何真正实现的。对于一些函数,可以声明和使用另一个临时的 Stack 对象)。

(1) 函数 **bool** full(Stack &s)没有改变 Stack s,并且根据 Stack s 是否已满而返回 true 或 false 值。

(2) 函数 Error_code pop_top(Stack &s, Stack_entry &t)从 Stack s 中删除栈顶元素,并且将它的值返回给输出参数 t。

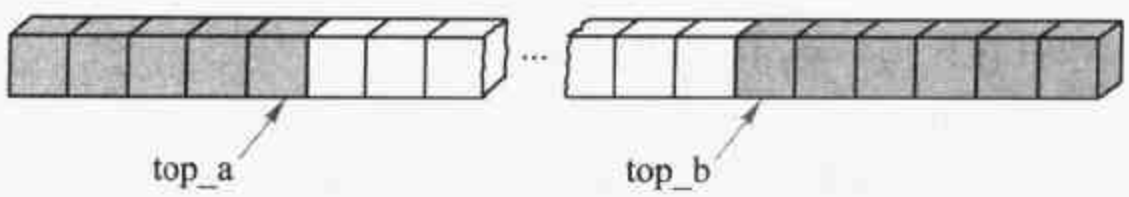
(3) 函数 **void** clear(Stack &s)删除所有的元素,并且返回空栈 s。

(4) 函数 **int** size(Stack &s)不改变栈 Stack s,并返回栈 Stack 中的元素个数。

(5) 函数 **void** delete_all(Stack &s, Stack_entry x)从 s 中删除所有的 x,并保持 s 中其他元素的相对次序不变。

*** 两个共存的栈

E4. 有时一个程序需要两个包含同样数据类型的栈。如果两个栈存放在不同的数组里,那么当一个栈溢出时,另一个栈可能还有相当多的未用空间。避免这个问题的一个简洁的方法是把所有的空间都放在一个数组里,让一个栈从数组的一端增长,另一个栈从数组的另一端以相反的方向朝第一个栈增长。以这种方式,如果一个栈大另一个栈小,则可以同时满足它们,除非所有的空间都被真正地使用了,否则,将不会产生溢出。定义一个新的类 Double_stack,包含(作为私有数据成员)数组和两个符号 top_a 和 top_b,并且为方法 Double_stack(),push_a(),push_b(),pop_a()和 pop_b()编写代码,处理一个 Double_stack 内的两个栈。



程序设计项目 2.2

P1. 汇编课本中的合适的声明,生成文件 stack.h 和 stack.c,并且验证 stack.c 能正确地编译,因此类 Stack 可以供以后的客户程序使用。

*** 质数约数

P2. 编写一个程序,它使用 Stack 读取一个整数并且以降序输出它的所有质数。如,整数 2100 的输出为:

7 5 5 3 2 2

(提示:任何大于 1 的整数的最小约数肯定是一个质数。)

① 友元函数可以访问C++类中的所有成员,即便是私有成员。

2.3 应用:桌面计算器

* * * 逆波兰计算

本节构造一个程序用于模拟简单计算器的行为,能进行加、减、乘、除及其他一些可能的操作。现在已有许多种计算器可用,因此我们的程序可以模拟其中的任何一个。然而,为了进一步说明栈的使用,我们选择模拟通常所谓的逆波兰计算器。在这样的计算器中,操作数(通常是数字)在操作被指定前输入,操作数放入栈中。当操作执行时,操作数从栈中弹出并且把操作结果入栈。

* * * 示例

* * * 不需要圆括号

我们用? 表示读取一个操作数并且将它入栈的指令;+,-、* 和/表示算术运算符;=表示打印栈顶(不是弹出)命令。此外,我们用 a,b,c 和 d 表示数值,如 3.14 或-7。指令? a? b + =意味着读取并且存储数字 a 和 b,计算并存储它们的和,然后打印出和。指令? a? b+? c? d * =需要 4 个数值操作数,并且打印的结果是(a+b) * (c+d)的值。类似地,指令? a? b? c-= *? d+=意味着把数值 a,b 和 c 入栈,用 b-c 代替 b 和 c 并且打印出其值,计算 a * (b - c),将 d 入栈,最后计算和打印(a * (b-c))+d。逆波兰计算器的一个优点是任何一个表达式,不管多复杂,都可以不用圆括号进行描述。

如果有 UNIX 系统,则可以用命令 dc 实验逆波兰计算器。

波兰符号对编译器来说和计算器一样有用,并且对它的研究形成了第 13 章的主题。然而,现在几分钟的逆波兰计算器的练习将使你对它的使用感到很轻松。

很明显,在逆波兰计算器里我们应当使用栈。在做出这个决定后,计算器程序的任务就变得很简单。主程序声明元素类型为 double 的栈,接受新的命令,并且只要希望就执行它们。

我们将在程序中运用泛型 Stack 实现,用 typedef 语句设置 Stack_entry 的类型,然后将它包含进 Stack 的定义文件 stack.h 中。

```
typedef double Stack_entry;
# include "stack.h"

int main()
/* Post: The program has executed simple arithmetic commands entered by the user.
  Uses: The class Stack and the functions introduction, instructions, do_command and,
        andget command . */
{
  Stack stored numbers;
  introduction();
  instructions();
  while (do_command(get_command(), stored_numbers));
}
```

辅助函数 get_command 从用户那里得到一个命令,检查其是否合法,并且用字符串函数 tolower()将它转化为小写,tolower()函数在标准头文件 cctype 中定义。(文件 cctype 或者其旧版 ctype. h,能通过我们的标准实用程序库自动包含,见附录 C。)

* * * 用户命令

为了实现 get_command，我们决定用字符?，=，+，-，* 和/表示用户能键入的命令，此处? 要求用户输入数值；=输出操作的结果；剩余的符号分别表示加、减、乘和除。

```
char get_command()
{
  char command;
  bool waiting=true;
  cout<<"Select command and press<Enter>:";
  while (waiting){
    cin>>command;
    command=tolower(command);
    if(command=='? ' || command=='=' ||command=='+' ||
      command=='-' ||command=='*' ||command=='/' ||
      command=='q')waiting=false;
    else {
      cout<<"Please enter a valid command:"<<endl
          <<"[?]push to stack[=]print top"<<endl
          <<"[+] [-][*] [/] are arithmetic operations"<<endl
          <<"[Q]uit."<<endl;
    }
  }
  return command;
}
```

* * * 执行用户命令

最后，选择和执行用户命令的工作是函数 do_command 的任务。这里我们给出了 do_command 函数的简要形式，其中仅仅在它的主要选择语句里编码了一些可能的命令。

```
bool do_command(char command, Stack &numbers)
/*Pre: The first parameter specifies a valid calculator command.
  Post: The command specified by the first parameter has been applied to the Stack of
        numbers given by the second parameter. A result of true is returned unless
        command=='q' .
  Uses: The class Stack. */
{
  double p, q;
  switch (command) {
  case '?':
    cout<<"Enter a real number:"<<flush;
    cin>>p;
    if(numbers.push(p)==overflow)
      cout<<"Warning:Stack full, lost number"<<endl;
    break;
  case '=':
    if(numbers.top(p)==underflow)
      cout<<"Stack empty"<<endl;
    else
      ccut<<p<<endl;
    break;
  case '+':
    if(numbers.top(p)==underflow)
      cout<<"Stack empty"<<endl;
```

```
    else {
      numbers.pop();
      if(numbers.top(q)==underflow) {
        cout<< "Stack has just one entry"<<endl;
        numbers.push(p);
      }
      else {
      numbers.pop();
      if(numbers.push(q+p)==overflow)
        cout<<"Warning: Stack full, lost result"<<endl;
      }
    }
    break;
  //  Add options for further user commands.
  case 'q':
    cout<<"Calculation finished.\n";
    return false;
  }
  return true;
}
```

当调用这个函数时，应当通过引用来传递 Stack 的参数，因为它的值可能需要被修改。比如，如果命令参数是＋，则我们通常从栈 Stack 中弹出两个值并且把它们的和入栈，这当然改变了 Stack。

函数 do_command 允许另外一个用户命令 q，即退出程序。

练习 2.3

E1. 如果在我们的计算器里使用了标准库的栈类，方法 top()返回栈顶元素作为结果，则函数 do_command 可以写成如下精简的形式：

```
case '-': numbers.push(numbers.pop() - numbers.pop());
```

(1) 假设这个语句能正确运行，解释为什么它将是一种不良的程序设计风格。

(2) 可能两个不同的C++编译器都严格遵守标准C++，这两个编译器对这条语句的翻译方式将导致程序运行时给出不同的答案。解释这种情况是如何发生的。

E2. 讨论使计算器能处理复数所需的步骤。

程序设计项目 2.3

P1. 编译本节开发的函数并且在代码里做一些必要的修改，以形成一个能运行的计算器程序。

P2. 编写一个函数，使它能够交换栈顶的两个数值，并且将此项功能作为一条新的命令包含进程序。

P3. 编写一个函数，使它可以累加栈中所有的数值，并且将此项功能作为一条新的命令包含进程序。

P4. 编写一个函数，使它可以对栈中所有的数值求平均值，并且将此项功能作为一条新的命

令包含进程序。

2.4 应用:括号的匹配

用C++语言设计的程序包含几种不同类型的括号,比如,用于括人表达式、函数变量、数组下标和代码块的括号。如我们所知,程序中使用的括号必须匹配。

比如,下面的字符串

```
{a=(1+v(b[3+c[4]]));
```

不可能括号匹配,因为它有5个开括号和仅仅4个关括号:像许多第一次设计的C++程序一样,漏掉了最后一个括号。字符串

```
{a=(b[0)+1];}
```

有同等数量的开括号和关括号,但我们能发现它有不匹配的括号,因为它的第一个关括号")"和最近的开括号"["不匹配。另外,括号序列

```
{()[()]}
```

虽然不是一个合法的C++程序的一部分,但它是匹配的。

* * *说明

这一节将实现程序来检查一个输人文本文件中的括号是否正确匹配。为简单起见,我们将注意力限制于{,},(,),[和]。而且,我们只是读入一行字符,并忽略掉其中所有非括号的字符。在检查括号的实际C++程序中,需要将特殊的规则应用于注释和字符串内的括号,并且还必须认识到符号"<"和">"也能用作括号(比如,在 stack<double>numbers 定义中,这是2.1.3节中的程序使用过的定义)。

* * *算法

如果将括号匹配的规则形式化,则可以很快得到下面的算法:逐个字符地读入程序设计文件,如果遇到开括号(,[和{,则认为它们是不匹配的,需要将它们保存直到找到一个匹配的括号。任何关括号),]和}在括号类型上必须与最后未匹配的开括号相对应,现在应重新检索此开括号并且将它从存储器中删除。最后,在程序末尾,必须检查是否还留下未被匹配的开括号。

* * *数据结构:栈

我们看到检测括号匹配的程序需要逐字符地处理输人文件,并且,在它处理输人的过程中,需要某种方式来记住现在未被匹配的任意括号。这些括号应当以与它们的输入顺序相反的顺序重新检索,因此 Stack 是用于存储它们的一种有吸引力的选择。

一旦决定使用栈,那么程序需要做的仅仅是在输人字符上循环,直到检测到一个括号错误或是输人文件结束。无论何时找到一个括号,就应用一个合适的 Stack 操作。因此我们得到下面的程序:

```
int main()
 *Post: The program has notified the user of any bracket mismatch in the standard input
        file.
```

```
  Uses: The class Stack. */
{
  Stack openings;
  char symbol;
  bool is_matched=true;
  while(is_matched && (symbol=cin.get()!='\n'){
    if(symbol=='{'||symbol=='('||symbol=='[')
      openings.push(symbol);
    if(symbol=='}'||symbol==')'||symbol==']'){
      if(openings.empty()){
        cout<<"Unmatched closing bracket"<<symbol
            <<"detected."<<endl;
        is_matched=false;
      }
      else {
        char match;
        openings.top(match);
        openings.pop();
        is_matched=(symbol=='}' && match=='{')
                   ||(symbol==')' && match=='(')
                   ||(symbol==']' && match=='[');
        if(!is_matched)
          cout<<"Bad match"<<match<<symbol<<endl;
      }
    }
  }
  if(!openings.empty())
    cout<<"Unmatched opening bracket(s)detected."<<endl;
}
```

程序设计项目 2.4

P1. 修改括号匹配程序以使它能够读入整个输入文件。

P2. 修改括号匹配程序以使输入字符回显输出，且在输出文件里标识出单个未匹配的关括号。

P3. 将C++的注释和字符串并入括号匹配程序，忽略掉注释或字符串内的任何括号。

2.5　抽象数据类型及其实现

2.5.1　引言

在栈的任何应用中，本来可以用数组和计数器代替栈，这样对每个栈操作需要用一组数组和计数器操作来代替。比如，括号检查程序可以使用如下的语句：

```
if(counter<max){
  openings[counter]=symbol;
  counter++;
}
```

在某些方面，这看起来像是一种简单的方法，因为代码直接易懂，它在许多方面都比设置一个类并定义它的所有方法更为简单。

然而，这种方法的一个主要的缺点是程序编写者（和任何阅读者）在每次使用栈的时候都必须花相当多的努力去验证数组的下标操作的细节，而不能将注意力集中在栈的实际使用方式上。这种不必要的努力直接导致程序设计者未能认识到栈的一般概念和给定应用所需要的特殊实现之间的区别。

另外一个应用可能包括下列指令，而不是简单的栈操作：

```
if((xxt==mxx) || (xxt>mxx))
    try_again();
else{
    xx[xxt]=wi;
    xxt++;
}
```

如果孤立地看，甚至看不出这段代码实质上与前一段有同样的功能，这两段代码的目的都是将一个元素放在栈顶。

* * * 类比

不同学科的研究者经常有基本类似的思想，但可能是为不同的目的而设计，或者采用不同的语言来表达。经常是过了多年以后，大家才认识到工作的类似性，但是一旦有了这种认识，来自于一个学科的知识往往能有助于其他的学科。虽然如此，在计算机科学领域，同样的基本思想经常以相当不同的伪装出现，从而使这种类似性也变得模糊。但是如果能够发现并强调这种类似性，则我们能总结出其中的思想并且获得更简单的方法去满足许多应用的需求。

* * * 实现

基本结构实现的方式对程序的设计和结果的性能及有用性将产生实质性的影响。有时这种影响可能是细微的，比如关于实数的基本的数学概念由计算机实现时，通常是（不总是）以一定精度的浮点数的形式，在其实现中的内在的限制经常因为舍入错误而带来困难。在数据的逻辑结构和它在计算机内存中的实现之间划出一个明显的界限将有助于我们设计程序。首先是识别数据间的逻辑连接并在逻辑数据结构内体现这种连接，随后就能考虑数据结构并确定实现它们的最佳方式，以期提高程序设计和运行的效率。通过将这些决策分离，从而使它们二者都变得更简单，而且也能避免出现过早进行约束的缺陷。

为了帮助阐明这种区别和实现更为普遍，现在我们以尽可能通用的观点来考虑数据结构。

2.5.2 一般定义

1. 数学概念

数学是归纳概括的精髓，因此它提供了定义所需要的语言。我们从定义类型（type）开始：

定义 类型是一个集合，集合的元素称为类型的值（value）。

因此，整数类型(integer)是所有整数的集合；实数类型(real)是所有实数的集合；字符类型(character)是算法所能够操作的符号的集合。

注意，我们总能区分抽象数据类型及其实现：比如C++类型 int 并不是所有整数的集合，它仅仅是由特定计算机里能够直接表示的那些整数组成，最大的整数依赖于该计算机的字长。类似地，C++的 float 和 double 类型通常是指浮点数(分开的尾数和指数)的集合，这里的浮点数仅仅是所有实数集合的一个小的子集。

2. 原子和结构化类型

像 int，float 和 char 这样的类型称为原子类型，因为我们认为它们的值仅是单个元素，不会对它进行再分。然而，计算机语言，如C++，提供了如数组、类和指针这些工具，可以用这些工具去建立新的类型，我们称之为结构化类型。结构化类型的单个的值(即此类型集合中的单个的元素)是一个结构化对象，如一个顺序栈。结构化类型的值包括两种成分：它由组元元素构成，并且有一个结构来提供将这些组元元素放在一起的规则集合。

*** 构造类型

从通用观点考虑，我们应当使用数学工具来提供用于构造结构化类型的规则，这些工具包括集合、序列和函数。而对各种各样的表的研究来说，所需的一种工具是有限序列，并且我们在对它的定义中使用了数学归纳法。① 归纳定义(就像归纳证明)包括两个部分：一是初始情况，二是依据前面情况做出的一般情况的定义。

定义　长度为 0 的序列是空的序列，由集合 T 中的元素构成的长度 $n \geqslant 1$ 的序列是一个有序对 (S_{n-1}, t)，其中 S_{n-1} 是由 T 中的元素构成的长度为 $n-1$ 的序列，而 t 是 T 的一个元素。

用这个定义，我们可以从空的序列开始，每次增加 T 中的一个新元素，从而能够建立越来越长的序列。

*** 连续与顺序的比较

从现在开始我们要在连续的(sequential)和顺序的(contiguous)这两个词之间做出一个仔细的区分。“连续”这个词意味着元素形成一个序列，而“顺序”这个词意味着元素在内存中有相邻的地址，因此我们能够在顺序实现中谈论连续表。

3. 抽象数据类型

*** 表的定义

有限序列的定义使我们马上就可能得到表的定义：项类型为 T 的表只是集合 T 的元素的有限序列。

接下来我们打算定义栈，但如果由你来考虑这个定义的话，你会意识到当考虑项的序列时，无法将它同来自表的这些结构加以区别。能做出修改或访问的操作或方法是栈的主要特征，因此我们在定义有限序列的同时，用结构化的规则包含有关这些操作的说明，得到如

① 参见附录 A 中利用归纳法证明的例子。

下定义：

定义 元素类型为 T 的栈是 T 的元素的一个有限序列，同时带有如下操作：

1. 创建一个栈，并使它为空(create)。
2. 测试栈是否为空(empty)。
3. 倘若栈不满，则将一个新元素压入栈顶(push)。
4. 倘若栈不空，则从栈顶弹出一个元素(pop)。
5. 倘若栈不空，则检索栈顶元素(top)。

＊＊＊抽象数据类型

注意此定义没有提到要实现的抽象数据类型栈的方式。在紧接下来的章节里，我们将研究栈的几种不同的实现方法，这个新的定义同样很好地适用于其中的任何一种实现。这个定义产生了所谓的抽象数据类型(Abstract Data Type)，简称为 ADT。任何抽象数据类型的定义都包括两个部分：第一是分量间彼此相关的方式的描述，第二是能对抽象数据类型的元素执行的操作的说明。

2.5.3 数据说明的细化

我们已经得到了抽象数据类型的通用的定义，现在开始对它进行更加详细的描述，因为所有工作的目标就是找到有助于设计程序的普遍原理，但为了实现这个目标，我们还需要更多的细节。

＊＊＊自顶向下的说明

实际上，算法的自顶向下的细化过程和我们现在正在讨论的数据结构的自顶向下的说明过程之间有极为相似之处。在算法设计中，我们由概括性的然而是精确的问题说明开始，然后逐步描述更多的细节，直到设计出完整的程序。在数据说明里，我们从问题要求的数学概念和抽象数据类型的选择开始，逐步说明更多的细节，直到最后能够用类来实现我们的数据结构。

＊＊＊细化阶段

在这个说明过程里所需要的细化阶段的数目依赖于应用程序。大型软件系统的设计比单一的小型软件程序需要更多的决策，并且这些决策应在几个不同的细化阶段做出。虽然不同的问题需要不同数目的细化阶段，而且有时这些阶段之间的边界是模糊的，但是我们能够选出 4 个不同层次的细化过程。

＊＊＊概念

1. 在抽象层，我们确定数据是如何彼此相关并且需要什么操作，但不关心数据的实际存储方式或者操作的实际步骤。

＊＊＊算法

2. 在数据结构层，我们描述足够的细节，从而能够分析方法的行为并且做出适合问题的恰当的选择。

＊＊＊程序设计

3. 在实现层，我们将确定如何在计算机内存中表示数据结构。

4. 在应用层，我们解决特定应用要求的所有细节，如变量命名或者是应用提出的特殊操作要求。

最初的两层经常称为概念层，因为在这些层次上更关心的是问题求解，而不是程序设计。中间两层可称为算法层，因为它们关心用于表示和操作数据的精确的方法。最后两层专门涉及程序设计。

C++里实现数据结构的任务是从概念信息开始的，此信息经常是一个 ADT 的定义，C++然后对它加以提炼并得到一个C++类。C++类的方法自然对应到 ADT 的操作，而C++类的数据成员则对应于我们选择去表示 ADT 的物理数据结构。这样，从抽象的 ADT 进到数据结构，然后继续进到实现的过程直接生成一个C++类的定义。

作为本节的总结，我们将重申最重要的原理并将它作为程序设计的技术规则：

程序设计技术规则

让数据组织程序，同时提炼算法和数据结构。

程序设计技术规则

一旦数据完全结构化，算法就几乎形成了。

练习 2.5

E1. 给出 2.2 节练习 E1 中扩展栈的形式化定义。

E2. 在数学中，集合 $T_1, T_2, \cdots, T_n$ 的笛卡尔积定义为所有 n -元组$(t_1, t_2, \cdots, t_n)$的集合，此处对所有的 $i, 1 \leqslant i \leqslant n, t_i$ 是 T_i 的一个成员。使用笛卡尔积给出一个类的精确定义。

启示和易犯的错误

1. 使用数据结构阐明程序的逻辑。
2. 在实现数据结构时使用信息隐藏和封装：使用函数访问数据结构，并且将它们留在类中并与应用程序分离。
3. 尽可能推迟实现数据结构细节的决定。
4. 栈是最简单的数据结构之一，需要时使用栈。
5. 在任何需要反转数据的问题里，考虑使用栈去存储数据。
6. 避免使用错综复杂的方法存储数据，错综复杂的方法通常不能推广到新的情形。
7. 确信初始化数据结构。
8. 设计算法时，总是小心极端情形并妥善地处理它们。跟踪算法以判断在极端情况下将会发生什么，特别是当数据结构为空或者满的时候。
9. 选择实现前，确信已在抽象层充分地描述所有的数据结构及其相关的操作。

复习题

2.1

1. 什么是标准库?
2. 什么是栈的方法?
3. 将数据结构的操作作为方法编写的好处是什么?

2.2

4. 信息隐藏和封装之间有何区别?
5. 描述可以被C++采用的3种不同的处理错误的方法。
6. 给出在C++中实现泛型数据结构的两种不同方法。

2.3

7. 使用逆波兰规范作为计算器的原因是什么?

2.5

8. 任何抽象数据类型的定义中必须包括哪两部分?
9. 在抽象数据类型里,有多少是用来描述实现的?
10. 例举(按从抽象到具体的次序)数据说明细化的4个层次。

进阶参考书目

＊＊＊栈

对于许多与栈这样的数据结构相关的主题,以下几本书是附加信息、历史记录和数学分析的最好资源,这些书几乎可以看作是它们各自所讨论的计算科学方面的百科全书:

＊＊＊百科全书式的参考文献:Knuth

Donald E. Knuth, *The Art of Computer Programming*, published Addison-Wesley, Reading, Mass.

迄今为止共有3卷:

1. *Fundamental Algorithms*, second edition, 1973, 634 pages.
2. *Seminumerical Algorithms*, second edition, 1980, 700 pages.
3. *Sorting and Searching*, 1973, 722 pages.

在以后的章节里,我们将经常引用这些书,为方便起见,将只指出名字Knuth和卷号、页号。书中的算法同时采用英语和汇编语言编写,Knuth并且计算了详细的操作数以比较不同的算法。

下面这本重要的参考书用大部分篇幅详细地描述了C++的标准库:Bjarne Stroustrup, *The C++ Programming Language*, third edition, Addison-Wesley, Reading, Mass., 1997.

波兰表示法是如此自然而有用,有人可能认为它的发现是在几百年前,但如果我们指出它是20世纪的发现,这可能会令你感到惊奇:Jan Łukasiewicz, *Elementy Logiki Matematyczny*, Warsaw, 1929; 英文译本: *Elements of Mathematical Logic*, Pergamon Press, 1963.

第3章　队　　列

队列这种数据结构模拟一排等待服务的人。和栈一样，队列也是一种最简单的数据结构。这一章详细描述队列的属性，研究如何应用它们，并且检验其不同的实现。队列的实现详细地说明了C++中派生类的使用和类继承这一重要的面向对象技术。

3.1　定义

在日常英语中，队列的定义是排队，就像一排人等着买票，队列中的第一个人将最先得到服务。对计算机应用来说，类似地，我们将队列定义为一个表，所有的增加都在表的一端进行，所有的删除都在表的另一端进行。队列也称为先进先出(first in, first out)表，或简称为 FIFO。



图 3.1　队列

＊＊＊应用

队列的应用甚至比栈的应用更加普遍，因为在用计算机执行任务时，就像生活中一样，经常需要等待轮次才能访问某个东西。在计算机系统里可能有任务队列在等待打印、等待磁盘访问或者是在多任务时等待使用 CPU。在单个程序里，一个队列可能有多个要求，或者一个任务可能创建其他的任务，因此必须将它们保存在一个队列中依次处理。

＊＊＊队头和队尾

队列里准备被服务的元素，也就是第一个将从队列中删除的元素称为队头(front，有时称为 head)。类似地，队列中的最后一个元素，即最近加入的一个元素称为队尾(rear 或 tail)。

3.1.1 队列操作

* * * 操作

为了完成队列的 ADT 定义，我们先说明它所允许的所有的操作。对每一种操作，我们将给出它的方法名并完成其说明的后置条件，当阅读这些说明时应注意到与栈相对应的操作的相似性。就像对栈的处理一样，我们将实现拥有泛型元素的队列，并称这种类型为 Queue_entry。

对任何队列要做的第一步工作是使用构造函数对它进行初始化以供将来使用：

```
Queue::Queue();
```
后置条件：建立队列并将其初始化为空。

接下来是队列基本操作的定义。

```
Error_code Queue::append(const Queue_entry &x);
```
后置条件：如果队列有空间，则将 x 加入到队尾，否则返回上溢错误码。

```
Error_code Queue::server();
```
后置条件：如果队列非空，则删除队头元素，否则返回下溢错误码。

```
Error_code Queue::retrieve(Queue_entry &x) const;
```
后置条件：如果队列非空，则队头元素记为 x，否则返回下溢错误码。

```
bool Queue::empty() const;
```
后置条件：如果队列为空则返回 true，否则返回 false。

* * * 可选的名称：insert，delete，enqueue 和 dequeue

将 append 和 serve 用于队列的基本操作清楚地表明要执行什么动作，而且避免了与我们在其他数据类型中使用的术语相混淆。然而也经常将其他的名称如术语 insert 和 delete 或铸造的词 enqueue 和 dequeue 等用于这些操作。

注意，任何试图向一个已满的队列中增加元素或者从一个已空的队列中删除元素的操作都将产生错误代码。因此我们的队列将使用和栈一样的枚举 Error_code 定义，包括代码 success，underflow，overflow。

Queue 方法的说明表明了我们的C++类的定义是基于如下的框架。

```
class Queue {
public:
  Queue();
  bool empty()const;
  Error_code append(const Queue_entry &x);
```

```
  Error_code serve();
  Error_code retrieve(Queue_entry &x) const;
//  Additional members will represent queue data.
};
```

标准的模板库为 queue 类提供了一个模板。我们所称的操作 empty，append，serve 和 retrieve 在标准库中分别称为 empty，push，pop 和 front。然而，由于这些操作与栈的那些操作在行为上非常不同，因此我们宁可使用能够突出这些差别的操作名。标准库队列的实现也提供了称为 back 和 size 的操作，分别用于检查队列里的最后元素（也就是，最近加入的元素）和总共的元素数。

3.1.2 扩展的队列操作

除了基本的方法 append，serve，retrieve 和 empty 外，其他的队列操作有时也很有用。比如，队列方法 full 用于检查队列是否完全满，这个方法有时可能会显得很方便。

＊＊＊重新初始化

＊＊＊队列长度

还有另外 3 个操作对队列也非常有用。第一个是 clear，它使已经创建的队列变空。第二个是函数 size，它返回队列中的元素个数。第三个是函数 serve_and_retrieve，它组合了 serve 和 retrieve 的功能。

我们可以选择这些函数作为基本 Queue 类的附加方法。然而，在面向对象的语言如 C++中，可以创建新的类去重用旧类的方法和实现。在这个例子中，我们将创建一个名为 Extended_queue 的新类，除了 Queue 的基本方法外，它还允许有新的方法。我们说 Extended_queue 类是由 Queue 类派生的类。

＊＊＊派生类

＊＊＊继承

派生类通过将新的方法加入到一个已经存在的类中，提供了一种定义类的简单方法。派生类重用基类成员的能力称为继承（inheritance），继承是面向对象程序设计的基本特征之一。

我们用一个层次图详细地说明 Queue 类和派生的 Extended_queue 类之间的关系，如图 3.2 的（a）部分所示，层次图中的箭头从派生类指向派生它的基类。图 3.2（b）部分说明派生类如何继承基类的方法，之后派生类也能包含附加的方法。

＊＊＊Extended_queue 类

我们在C++中使用操作符：（冒号）定义派生类。类 Extended_queue 的定义如下：

```
class Extended_queue: public Queue {
public:
  bool full() const;
  int size() const;
  void clear();
  Error_code serve_and_retrieve(Queue_entry &item);
};
```

类定义第一行中的关键字 public 说明 Extended_queue 每个继承的成员与 Queue 的成

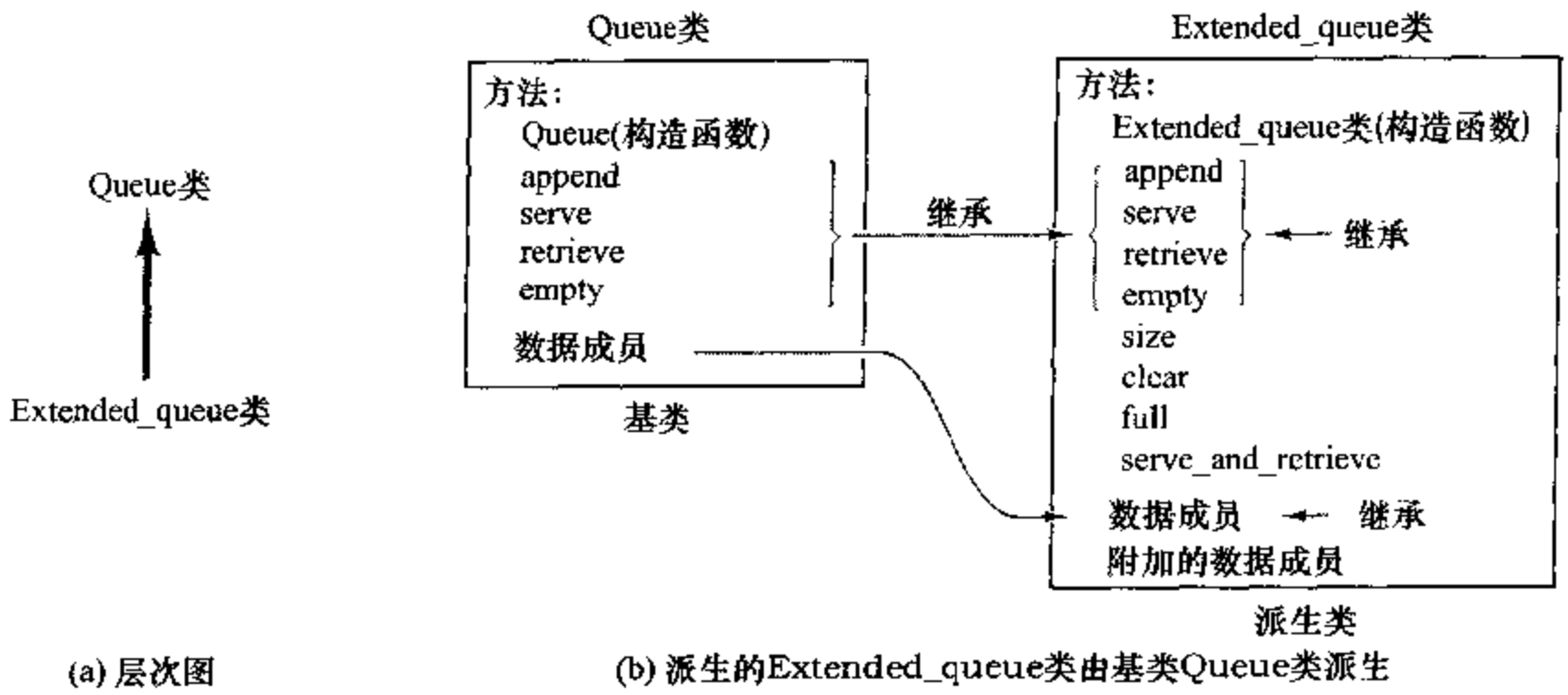


图 3.2 继承和派生类

员对客户程序具有同样的可见性。

用于 Extended_queue 类的新操作有下列说明。

* * *状态

```
bool Extended_queue::full() const;
```
后置条件:如果 Extended_queue 已满,则返回 true,否则返回 false。

* * *其他操作

```
void Extended_queue::clear();
```
后置条件:删除 Extended_queue 中的所有元素,使它为空。

```
int Extended_queue::size() const;
```
后置条件:返回 Extended_queue 里的元素个数。

```
Error_code Extended_queue::serve_and_retrieve(Queue_entry &item);
```
后置条件:如果 Extended_queue 为空则返回 underflow(下溢),否则删除并复制 Extended_queue 队头中的项到 item,并返回 success(成功)。

* * *is-a 关系

Extended_queue 类和 Queue 类之间的关系通常称为 is-a 关系,这是因为每个 Extended_queue 对象"是一个(is a)" Queue 对象,并具有其他的特征——即方法 serve_and_retrieve,full,size 和 clear。只要两个 ADT A 和 B 之间关系的口头描述包含短语"每个 A 是一个 B",我们就把代表 A 的类的实现看作是从代表 B 的类派生的过程。

* * * 例子

* * * has-a 关系

* * * 层次化

作为类间的 is-a 关系的另一个例子，考虑管理一个大学的预算的程序中所使用的C++类，其中的一些类是 University，Student，University_president 和 Person。每个学生是一个人，因此我们将 Student 类作为 Person 类的派生类创建，以此反映相关概念间的 is-a 关系。University_president 类也能作为 Person 类的派生类实现，它反映了另一个明显的 is-a 关系。University 类和 University_president 虽然有关，但并未反映出 is-a 关系，因为每一个大学有一个校长。我们说这两个类反映了一种"has-a"的关系。在实现中我们通过对类进行层次化，使得这种关系变得清晰，也就是说，通过在 University 类的定义里包含 University_president 类的数据成员。

练习 3.1

E1. 假设 q 是一个保存字符的 Queue，x 和 y 是字符变量。说明下列代码段中实行每一步骤后 q 的内容。

(1)
```
Queue q;
q.append('a');
q.serve();
q.append('b');
q.serve();
q.append('c');
q.append('d');
q.serve();
```

(2)
```
Queue q;
q.append('a');
q.append('b');
q.retrieve(x);
q.serve();
q.append('c');
q.append(x);
q.serve();
q.serve();
```

(3)
```
Queue q;
q.append('a');
x= 'b'
q.append('x');
q.retrieve(y);
q.serve();
q.append(x);
q.serve();
q.append(y);
```

E2. (计账)假设你是一个金融家，你在每年的一月、四月和九月购买 X 公司 100 股股票，每年六月和十一月卖掉 100 股。这些月份中每一股的价格是：

Jan	Apr	Jun	Sep	Nov
$10	$30	$20	$50	$30

确定你的资金收益或损失总量.使用下列数据结构进行计算：(1)FIFO(first-in，first out)；(2)LIFO(last-in，first－out)(也就是，假设你把股票交易证书放在(1)队列或(2)栈中)。你在本年结束时还拥有的 100 股不要参与计算。

E3. 使用课本中为栈和队列设计的方法编写函数完成下列任务。在编写每一个函数的过程中，确信在适当的时候检查空和满的结构。程序可以在需要时声明其他的局部结构。

(1) 将 Stack 中的所有元素移到 Queue。

(2) 将 Queue 中的所有元素移到 Stack。

(3) 将一个 Stack 置空，将它的元素放到另一个 Stack 的栈顶，且第一个 Stack 中的元素在另一个 Stack 中保持同样的相对次序。

(4) 将一个 Stack 置空，将它的元素放到另一个 Stack 的栈顶，且第一个 Stack 中的元素在另一个 Stack 中保持与原来相反的次序。

(5) 使用局部的 Stack 反转一个 Queue 中的所有元素的次序。

(6) 使用局部的 Queue 反转一个 Stack 中的所有元素的次序。

3.2 队列的实现

我们已经考虑了队列及其方法的定义，现在让我们改变角度，考虑如何利用计算机存储并作为一个C++类来实现队列。

1. 物理模型

* * * 缺点：多次移动

像我们对栈的处理一样，创建一个队列的简单方法是建立一个普通的数组去存放队列的元素，但现在必须跟踪队列的头和尾。一种策略是保持队列的头总是在数组的第一个位置，然后可以简单地在队尾追加元素，就好像向栈中增加元素一样，为此只需要增加用来指示队尾的计数器。然而这样会使得从队列删除一个元素的代价很大，因为当第一个元素被删除以后，所有剩余的元素都需要向前移动一个位置以填充空位。对较长的队列，这个过程将非常缓慢。因此虽然这种存储方法形象地模拟了等待被服务的一队人，但在计算机里它是一种低劣的选择。

2. 线性实现

* * * 缺点：报废的空间

因此为了有效地处理队列，需要两个指针，以便能同时跟踪队列的头和尾而不需移动任何元素。为了向队列中追加一个元素，只需将队尾加 1 并将元素放在那个位置；为了给一个元素提供服务，只需在队头位置获取它并将队头加 1。然而这种方法有一个主要缺陷，即队头和队尾都在增加而从未减少。如果操作如下，则即使队列中从不多于两个元素，也需要一个无界的存储空间：

append，append，serve，append，serve，append，serve，append，…

当然，问题在于当队列沿着数组向下移动时，数组开始处的存储空间被报废并且不会再使用。也许能将队列比作一条可以伸展的蛇沿着存储器爬行，蛇有时伸长，有时缩短，但如果它总是保持沿直线爬行，那么它很快能到达存储空间的尾部。

* * * 优点

然而，注意到对于队列规律性置空的应用(如允许建立一系列的请求直到某一个点，然后会有任务被初始化，并在其返回前清除所有的请求)，那么当队列为空的时刻，队头和队尾都能重置到数组的开头，使用二个指针的简单方案和直线存储就变成一种很有效的实现方式。

3. 循环数组

在概念上，我们只需要将数组想成是一个环而不是一条直线，以此克服空间的无效利用，见图 3.3。这样，当加入和删除队列中的元素时，队头将沿着数组继续追赶队尾，因此蛇将保持不定的爬行，但会停留在一个受限的环里。不同的时刻，队列将占据数组的不同部分，但我们决不必担心空间会用完，除非整个数组都被完全占用，但这种情况确实是算作溢出了。

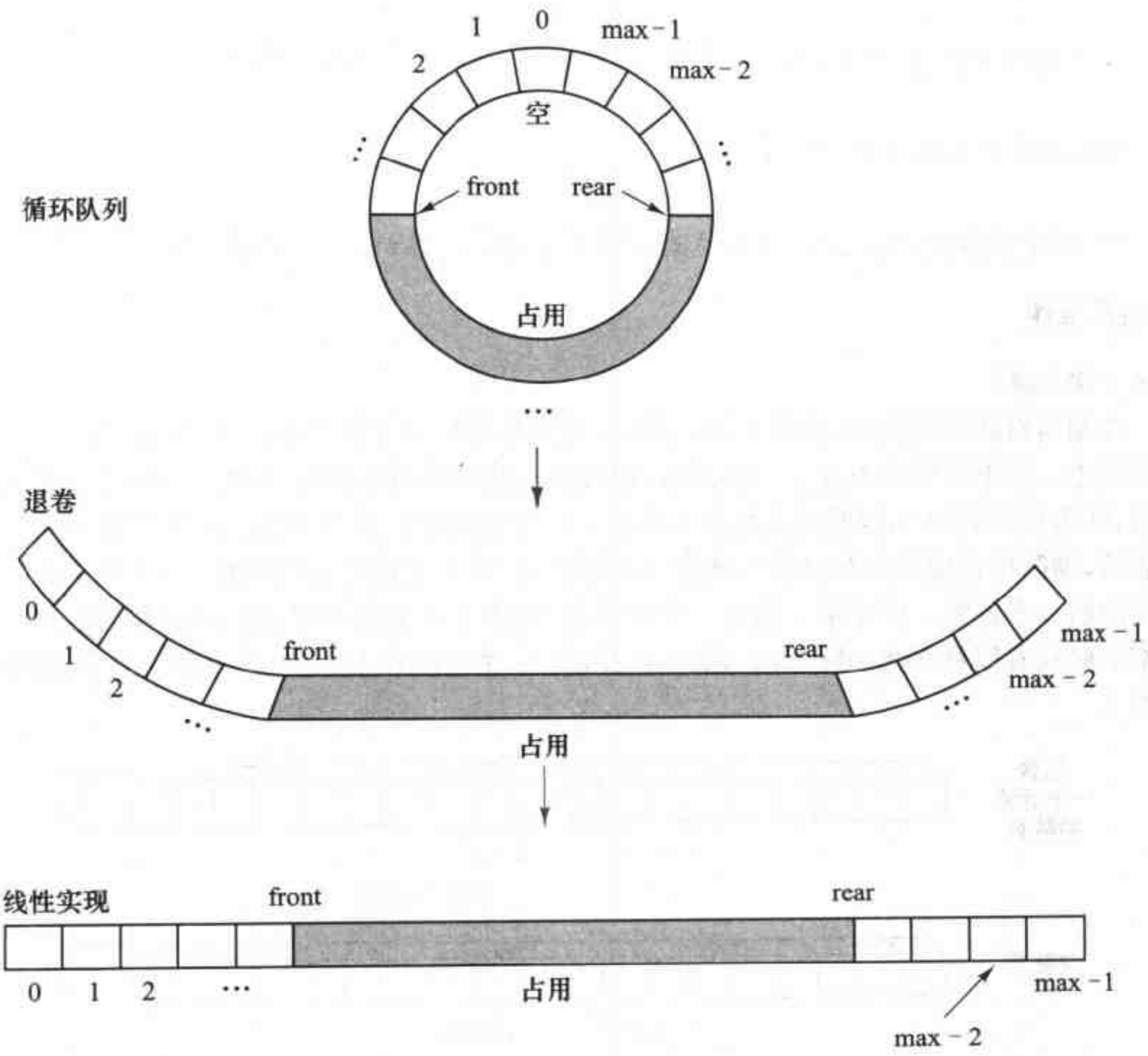


图 3.3 循环数组里的队列

4. 循环数组的实现

* * * 模运算

下一个问题是用一个普通的线性(即直线型的)数组实现一个循环数组。为了解决这个问题,我们将数组的编号设置为 0 到 max-1, max 是循环数组里总的元素个数。为了实现此循环数组,我们使用与线性数组同样的编号,移动指针就好像是做模运算一样:当指针加 1 后超过 max-1 时,则又从 0 开始。这就好像在圆钟上做运算一样,时钟的编号是从 1 到 12,如果在 10 点时再加 4 个小时,则得到 2 点。

这种线性表示方式的一种很粗糙的类比是牧师向跪在教堂前的人们布道,领圣餐者直到牧师走过去并得到服务后才移动。当牧师到达行的尾部时,他又返回到行头重新开始,因为这次是面对走向前的新的一行人。

5. C++里的循环数组

* * * 三元操作符?:

在C++中,我们可以如下使用三元操作符?:来增加循环数组的下标 i:

```
i=((i+1)==max) ? 0 : (i+1);
```

这种在C++里很少见的三元操作符?:的使用意义与下列表达式相同：

```
if((i+1)==max)i=0; else i=i+1;
```

或者我们可以用模操作符来写：

```
i=(i+1)% max
```

（应该进行检查，验证后一个表达式的结果总是在0到max - 1之间。）

6. 边界条件

* * * 空或满?

在编写对队列进行加、减的正式算法前，我们先考虑一下边界条件，也就是队列为空或满的标志。如果队列中正好有一个元素，那么队头指针将等于队尾指针。当这个元素被删除后，队头指针要加1，因此当队尾在队头前一个位置时表示队列为空。现在假设队列几乎是满的，则队尾远离队头并自始自终是围着环转，然后当数组已满的时候，队尾将正好位于队头的前一个位置。由此带来的另一个困难是：队头和队尾指针对于队列空和满这两种情况都恰好具有同样的相对位置！所以单看指针，并不能区分队列空和满。这种情形如图3.4。

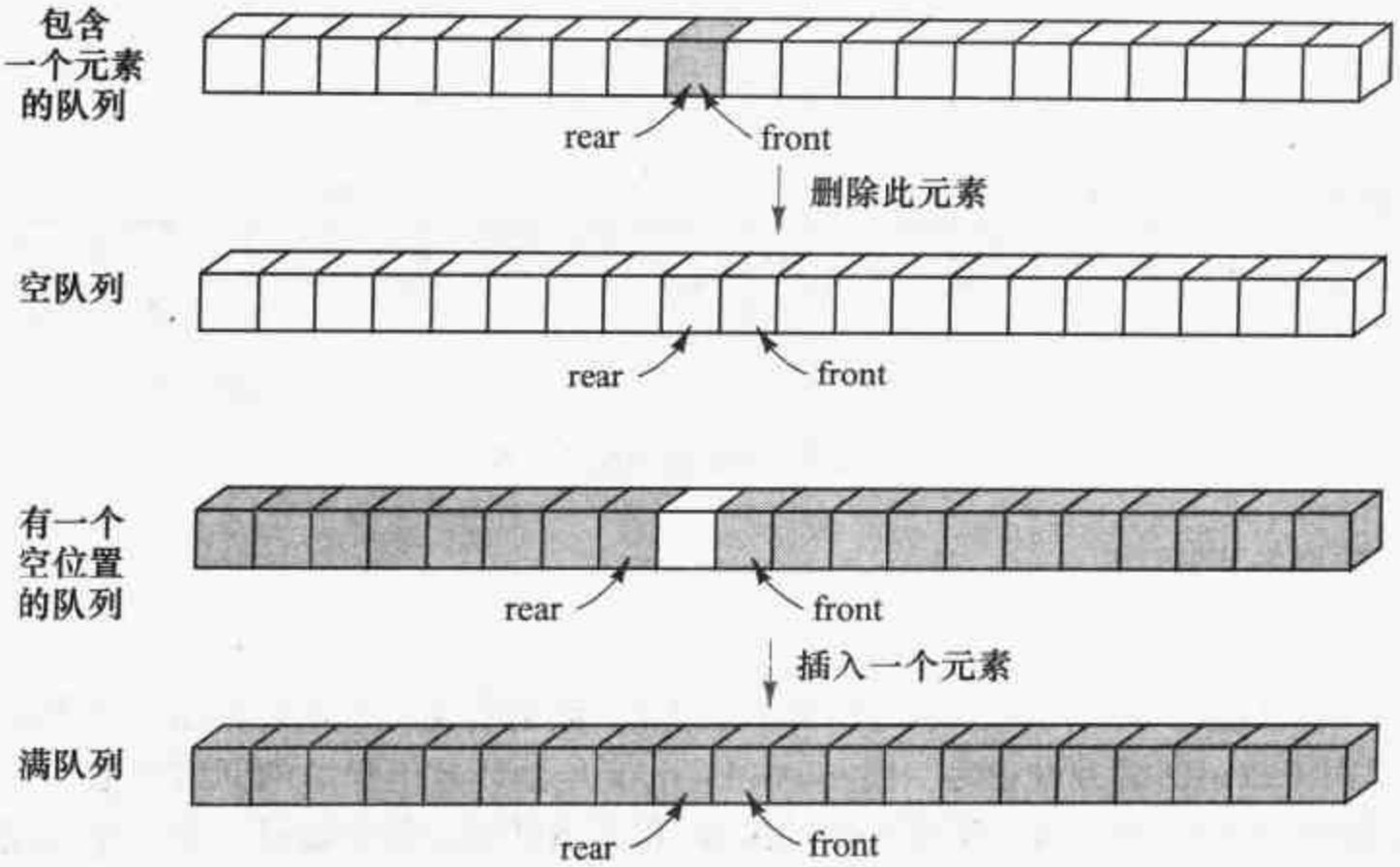


图3.4 空和满的队列

7. 可能的解决方案

* * * (1)空位

* * * (2)标志

* * * (3)特殊值

至少有3种本质上不同的方法可以解决这个问题。第一种方法是坚持在数组中留一个空位，当队尾指针已经移到队头指针的两个位置之内时，认为队列已满。第二种方法是引进

一个新的变量，它可以是布尔型，当队尾正好在队头前面时，指示队列已满(正如用于指示空的标志)，或者也可以使用一个整型变量来计算队列中元素的个数。第三种方法是将一个或两个指针设置成某个(某些)不会出现的值以指示空或满的队列，如可以将尾指针设置成-1以指示空队列。

8. 对实现的总结

为了总结关于队列的讨论，在此列出已经讨论过的实现队列的所有方法。

- 物理模型：线性数组的头总是在第一个位置，只要队头被删掉，所有的元素都要沿着数组移动。一般来说，在计算机中这是一种低劣的方法。
- 具有两个指针的线性数组总是在增长，如果队列能立即置空，那么这是一种好的方法。
- 循环数组带有头指针和尾指针，并且留有一个空位。
- 循环数组带有头指针和尾指针，并且有一个标志指示满(或空)。
- 循环数组带有头指针和尾指针，并且用整型变量对元素计数。
- 循环数组带有头指针和尾指针，并且采用特殊的值指示空。

* * * 推迟实现决策

在下一章中，我们要考虑另一种实现队列的方法，即采用链式结构。在这种实现表中要牢记的一件重要事情就是，如果在实现中有多种变化，那么我们总是应该将涉及数据结构如队列的使用问题同涉及其实现的问题分开，并且在程序设计时，某一时刻总是应该只考虑其中的一类问题。在考虑了如何在应用中使用队列，并且在设计完应用队列的客户程序代码之后，会有更多的信息来帮助选择适合于应用的队列的最佳实现。

> **程序设计技术规则**
>
> 使用信息隐藏：将数据结构的应用和它们的实现分离。

3.3 C++队列的循环实现

下面，我们编写正式的方法来实现队列和扩展的队列。从上一节可以明显看出可能有许多种实现，其中的一些实现与另一些实现之间只有很少的差别。因此，我们仅集中讨论其中一种实现，并将其他的留作练习。

使用计数器跟踪队列中的元素数的循环数组实现方法不仅说明了处理循环数组的技巧，也简化了扩展队列的某些操作的程序设计，因此我们只针对这种实现来讨论。

我们将队列看作存储在下标范围为0到(maxqueue-1)的数组中，它还包含类型为Queue_entry的元素。Queue数据成员front和rear将记录恰当的数组下标，数据成员count用于跟踪Queue中的元素数。Queue的类定义采用以下形式：

* * * Queue类的定义

```
const int maxqueue=10;      //  small value for testing

class Queue {
```

```
public:
  Queue();
  bool empty()const;
  Error_code serve();
  Error_code append(const Queue_entry &item);
  Error_code retrieve(Queue_entry &item)const;
protected:
  int count;
  int front, rear;
  Queue_entry entry[maxqueue];
};
```

* * * 受保护的可见性

注意到我们为 Qucue 的数据成员给出的是受保护的(protected)而不是私有的(private)可见性。对于客户程序代码,受保护的可见性与私有的可见性具有同样的意义,因此 Queue 类仍是被封装的。然而,派生类的成员函数允许访问基类的受保护的成员。因此,当为派生的 Extended_queue 类编写方法时,代码要能够使用 Queue 类的数据成员。如果缺乏这种访问,则 Extended_queue 的一些方法的实现将很低效。3.1.2 节已给出了扩展队列的类说明。

我们从初始化开始编写 Queue 方法的代码。

```
Queue::Queue()
/*Post: The Queue is initialized to be empty. */
{
  count=0;
  rear=maxqueue-1;
  front=0;
}

bool Queue::empty()const
/*Post: Return true if the Queue is empty, otherwise return false. */
{
  return count==0;
}
```

队列 Queue 中增加或删除的方法将严格遵循先前的讨论,注意必要时返回一个 Error_code。

```
Error_code Queue::append(const Queue_entry &item)
/*Post: item is added to the rear of the Queue. If the Queue is full return an Error_
        code of overflow and leave the Queue unchanged. */
{
  if (count>=maxqueue)return overflow;
  count++;
  rear=((rear+1)==maxqueue)? 0:(rear+1);
  entry[rear]= item;
  return success;
}

Error_code Queue::serve()
/*Post: The front of the Queue is removed. If the Queue is empty return an Error_code of
        underflow. */
{
  if (count<=0)return underflow;
```

```
  count--;
  front=((front+1)==maxqueue)? 0:(front+1);
  return success;
}

Error_code Queue::retrieve(Queue_entry &item)const
/*Post: The front of the Queue retrieved to the output parameter item. If the Queue is
        empty return an Error_code of underflow. */
{
  if (count<=0)return underflow;
  item=entry[front];
  return success;
}
```

我们将方法 empty 和 retrieve 留作练习，而考虑扩展队列的一种方法，这个方法给出扩展队列的长度，并且在实现中很容易编写。

```
int Extended_queue::size()const
/* Post: Return the number of entries in the Extended_queue.*/
{
  return count;
}
```

注意在编写方法 size 时，使用了受保护的 Queue 成员 count。如果 Queue 类中的数据成员有私有的可见性，那么它们对这个函数是不可用的，而我们关于 size 方法的代码需要组合调用公有的 Queue 方法 serve，retrieve 和 append。其他的 Extended_queue 方法 full，clear 和 serve_and_retrieve 都有类似的实现，我们把这些留作练习。

练习 3.3

E1. 写出本节中所实现的队列的遗留的方法：

(1) empty (2) retrieve

E2. 写出本节中所实现的扩展队列的遗留的方法：

(1) full (2) clear (3) serve_and_retrieve

E3. 编写队列在线性数组中的实现所需要的方法，可以假设必要时能够置空队列。

E4. 编写方法以如下方式实现队列：采用保持队列的头指针始终是在线性数组的第一个位置这种简单但缓慢的技术。

E5. 编写方法以如下方式在线性数组中实现队列：采用两个指针 front 和 rear，当尾指针到达数组的尾部时，将所有的元素移到数组的头部。

E6. 编写方法以如下方式实现队列：实现中不保持队列中的元素计数，但使用特殊的条件去指示队列为空：

$$\text{rear}=-1 \text{ 且 } \text{front}=0$$

E7. 重新编写课本中队列处理的方法，使用一个标志而不是保持队列中元素个数的计数来指示满队列。

E8. 编写用循环数组实现队列的方法，数组里有一个元素未用。也就是说，当尾指针在头指针之前两个位置时，我们认为队列是满的；当尾指针是在一个位置之前时，认为队列是

空的。

* * * 双端队列

词 deque(发音为“deck”或“DQ”)是 double-ended queue 的缩写,它表示一个表,其元素可以从队列的头部或尾部加入或删除,但是不改变表中的任何其他位置,因此双端队列同时是栈和队列的广义化。双端队列上的基本操作有:append_front,append_rear,serve_front,serve_rear,retrieve_front 和 retrieve_rear。

E9. 写出用线性数组实现双端队列的类定义和所需要的方法实现。

E10. 写出用循环数组实现双端队列所需要的方法,把 Deque 类看作是从 Queue 类派生的。(你能对用户隐藏 Queue 的方法吗?)

E11. 是使用线性数组还是使用循环数组实现双端队列更合适?为什么?

E12. 从图 2.3 可以看出,栈可以图形化地表示为笔直铁路线上的支路,当然,队列也能简单地表示成一条直路。设计并画出表示一个双端队列的铁路交换网,网络应该只有一个入口和一个出口。

E13. 假设编号为 1,2,3,4,5,6 的数据项以这种次序存在于输入流中。也就是说,第一个是 1,然后是 2,以此类推。通过使用(1)队列和(2)双端队列,是否能得到下列重新排列的输出?假设元素也以从左至右的顺序离开双端队列。

(1) 1 2 3 4 5 6　　(2) 2 4 3 6 5 1　　(3) 1 5 2 4 3 6

(4) 4 2 1 3 5 6　　(5) 1 2 6 4 5 3　　(6) 5 2 6 3 4 1

程序设计项目 3.3

P1. 编写一个函数,从终端读取一行输入。假设输入由两部分组成,之间用冒号‘:’分开。作为结果,函数应该产生如下的单个字符:

N　行中没有冒号。

L　左边部分(冒号前)比右边部分长。

R　右边部分(冒号后)比左边部分长。

D　左边部分和右边部分不同但有同样的长度。

S　左边部分和右边部分完全相同。

例如:

输入	输出
Sample Sample	N
Left:Right	R
Sample:Sample	S

在读取右边部分时,使用队列或扩展队列跟踪行的左边部分。

3.4 演示和测试

* * * 菜单驱动演示

在我们已经编写了一堆处理数据结构的方法和函数之后,应立即测试实现以确信它的每一部分都能正确工作。这样做最简单的方法之一是编写一个菜单驱动演示程序,它将创

建数据结构并允许用户以任何希望的次序，对数据结构执行所有可能的操作，只要用户希望，就可以打印出结果。现在让我们为扩展队列设计这样的程序，这个程序将作为基础，以供我们学习本书中更复杂的数据结构。

扩展队列中的元素可以是我们希望的任何类型，因此为了简单起见，我们使用字符队列，其元素是单个的字母、数字、标点符号等。

在主程序每一次迭代时，演示程序会要求用户选择一种操作，(如果可能)则在数据结构上执行那个操作并打印出结果。

因此主程序为：

```
int main()
/*Post: Accepts commands from user as a menu driven demonstration program for the class
        Extended_queue.
  Uses: The class Extended_queue and the functions introduction, get_command, and do_
        command.*/

{
  Extended_queue test_queue;
  introduction();
  while (do_command(get_command(), test_queue));
}
```

在这个演示程序中，用户将输入单个字符以选择一个命令。help 函数给出了各个命令的含义以及相应的字符，它本身可以被其他合适的命令激活：

```
void help()
/*Post: A help screen for the program is printed, giving the meaning of each command that
        the user may enter. */
{
  cout<<endl
    <<"This program allows the user to enter one command"<<endl
    <<"(but only one) on each input line."<<endl
    <<"For example, if the command S is entered, then"<<endl
    <<"the program will serve the front of the queue."<<endl
    <<endl
    <<"The valid commands are:"<<endl
    <<"A-Append the next input character to the extended queue"<<endl
    <<"S-Serve the front of the extended queue"<<endl
    <<"R-Retrieve and print the front entry."<<endl
    <<"#  -The current size of the extended queue"<<endl
    <<"C-Clear the extended queue (same as delete)"<<endl
    <<"P-Print the extended queue"<<endl
    <<"H-This help screen"<<endl
    <<"Q-Quit"<<endl
    <<"Press<Enter>to continue."<<flush;

  char c;
  do {
    cin.get(c);
  } while (c!='\n');
}
```

introduction 函数仅在程序启动时被激活一次，这个函数的目的是简短地解释程序做什

么，并引导用户如何开始。用户可以使用的进一步介绍将来自于 help 或 get_command。

函数 get_command 打印出菜单并从用户那里获取一个命令。这个函数与 2.3 节相应的函数相比有少许变化，我们将它的实现作为一个设计项目。

最后，选择和执行命令是函数 do_command 的任务，这个函数仅运行 switch 语句的合适的情况，我们给出此函数中包含 switch 语句的简化形式的部分版本。

```
bool do_command(char c, Extended_queue &test_queue)
/*Pre: c represents a valid command.
  Post: Performs the given command c on the Extended_queue test_queue. Returns false if
        c=='q', otherwise returns true.
  Uses: The class Extended_queue. */
{
  bool continue_input=true;
  Queue_entry x;
  switch (c) {
  case 'r':
    if(test_queue.retrieve(x)==underflow)
      cout<<"Queue is empty."<<endl;
    else
      cout<<endl
          <<"The first entry is:"<<x
          <<endl;
    break;
  case 'q':
    cout<<"Extended queue demonstration finished."<<endl;
    continue_input=false;
    break;
  // Additional cases will cover other commands.
  }
  return continue_input;
}
```

注意，我们在所有的测试函数中，已经仔细地维护了数据抽象的原理。Extended_queue 的说明和方法是在文件中，因此只要我们希望，就可以用另一种实现去代替特定的 Extended_queue 实现，而程序本身不用进一步的修改即能运行。

我们也编写了测试函数，因此随后可以使用这个程序去测试其他的数据结构，除了合法的操作和 introduction 及 help 窗口外，几乎不需要再做任何修改。

程序项目 3.4

P1. 通过实现函数 get_command 并完成 do_command 函数，完成菜单驱动演示程序，用于处理字符型元素的 Extended_queue。

P2. 类似于 Extended_queue 演示程序，编写一个菜单驱动演示程序用于处理双端队列的字符。

3.5 队列的应用:模拟

3.5.1 介绍

* * * 模拟

模拟是用一个系统去模仿另一个系统的行为。当在实用系统上做实验代价太大或者具有危险的时候,经常会用到模拟。物理模拟的例子如将风洞用于汽车车体的设计实验,用飞行模拟器训练航线飞行员等。数学模拟是用于描述某些系统的方程式系统,计算机模拟则是用程序的步骤模拟正在研究的系统行为。

* * * 计算机模拟

在计算机模拟中,研究的对象通常表示为数据,它经常作为由类给出的数据结构,类的成员描述了对象的特性,类的方法表示被研究的操作,描述这些操作的规则被翻译成计算机算法。通过改变数据的值或通过修改这些算法,可以在计算机模拟中观察到一些变化,然后据此得出关于实际系统行为的有价值的推论。

当系统中的一个对象涉及某个操作时,其他的对象和操作将经常需要保持等待,因此队列是计算机模拟中使用的重要数据结构。我们将研究最普通和最有用的一种计算机模拟,这种模拟主要是用队列作为它的基本数据结构。这些模拟模仿了系统(实际上经常称为队列系统)的行为,在这些系统里有成队的对象等待各种进程提供服务。

3.5.2 飞机场的模拟

作为一个具体的例子,考虑一个很繁忙的小型飞机场,这个飞机场只有一条飞机跑道(见图 3.5)。在每个单位时间内,只有一架飞机可以着陆,或者只有一架飞机可以起飞,但不允许同时着陆和起飞。飞机的到达和起飞是随机的,因此在任何给定的时刻,飞机跑道可能空闲,或者可能有一架飞机正在着陆或起飞,并且可能有若干架飞机在等待着陆或等待起飞。

* * * Plane 类

为了模拟飞机场,有必要创建一个 Plane 类,它的对象表示单个飞机。这个类显然需要一个初始化方法和表示起飞和着陆的方法。而且,在编写模拟的主程序时,会明显需要其他的 Plane 方法。我们也将使用一个 Runway 类存放关于飞机跑道的状态和操作的信息。这个类将维护等待着陆和起飞的飞机队列的成员。

* * * Random 类

在我们的模拟中,需要另一个 Random 类,它封装到达或离开跑道的飞机的随机特征,我们将在 3.5.3 节详细讨论这个类。在主程序里,我们将使用来自于 Random 类的单个方法 poisson,这个方法使用一个浮点型的参数(表示一个平均结果)并返回一个整型值。虽然返回的值是随机的,但在多次重复调用这个方法的过程中,它所具有的特征是返回值的平均值将匹配指定的参数。

在我们的模拟里,将特别关心飞机在起飞或着陆前需要在队列里等待的时间。因此,时

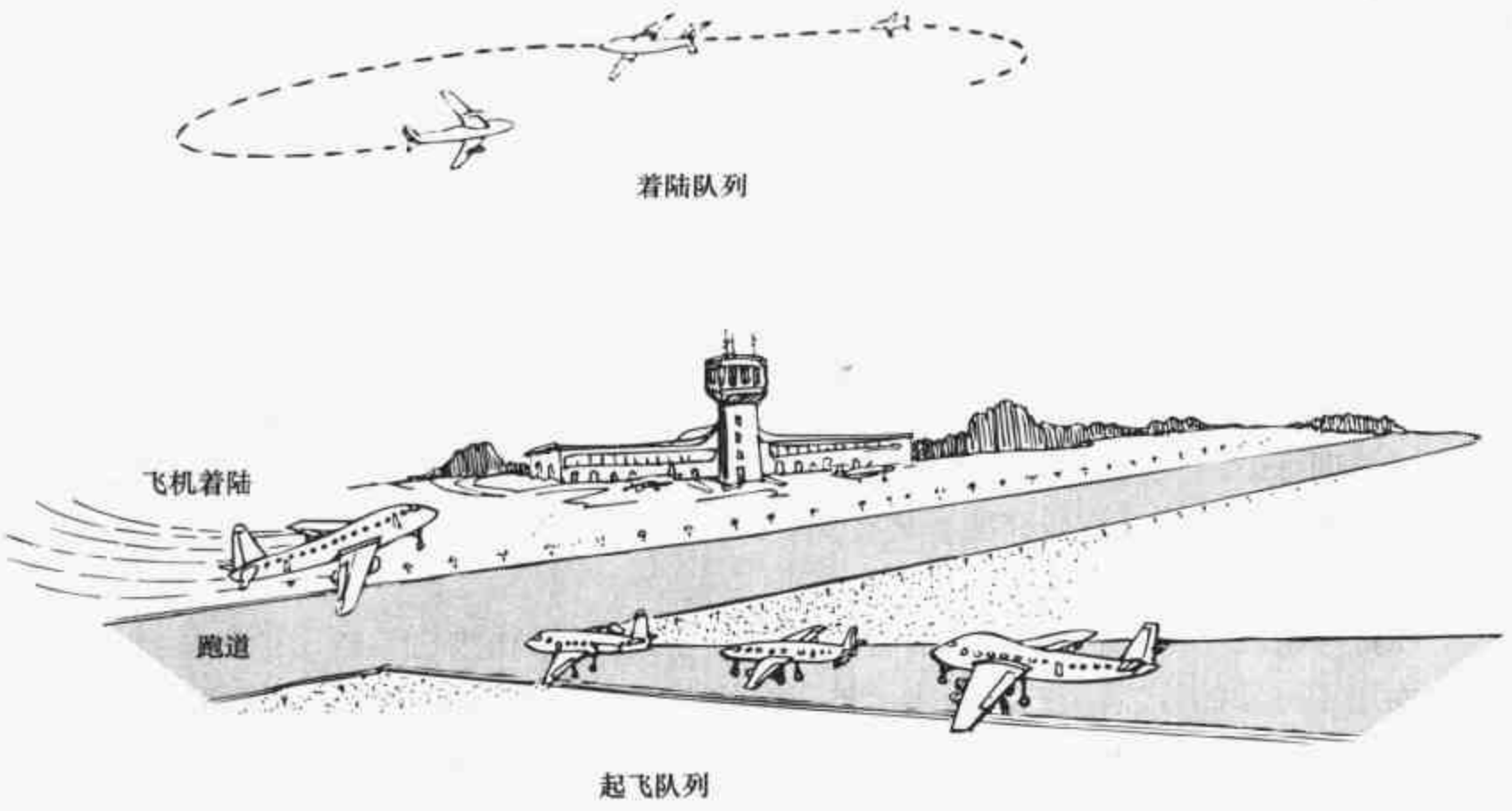


图 3.5 飞机场

间的度量对我们的程序最为重要。我们将模拟期间的时间分割成单元，使得在任一给定的时间单元内，仅有一架飞机能利用飞机跑道着陆或起飞。

如何处理着陆和起飞队列的精确细节将在设计 Runway 类时处理。类似地，主程序并不需要描述 Plane 的操作的精确方法。

```
int main()                    // Airport simulation program
/*Pre: The user must supply the number of time intervals the simulation is to run, the
       expected number of planes arriving, the expected number of planes departing per
       time interval, and the maximum allowed size for runway queues.
  Post: The program performs a random simulation of the airport, showing the status of
       the runway at each time interval, and prints out a summary of airport operation
       at the conclusion.
  Uses: Classes Runway , Plane , Random and functions run_idle , initialize . */
{
  int end_time;          //  time to run simulation
  int queue_limit;       //  size of Runway queues
  int flight_number=0;
  double arrival_rate, departure_rate;
  initialize(end_time, queue_limit, arrival_rate, departure_rate);
  Random variable;
  Runway small_airport(queue_limit);
  for (int current_time=0; current_time<end_time; current_time++){
                         //  loop over time intervals
    int number_arrivals=variable.poisson(arrival_rate);
                         //  current arrival requests
    for (int i=0; i<number_arrivals; i++){
      Plane current_plane(flight_number++, current_time, arriving);
      if(small_airport.can_land(current_plane)!=success)
        current_plane.refuse();
    }
```

```
    int number_departures=variable.poisson(departure_rate);
                              //  current departure requests
    for (int j=0; j<number_departures; j++){
      Plane current_plane(flight_number++, current_time, departing);
      if(small_airport.can_depart(current_plane)!=success)
        current_plane.refuse();
    }
    Plane moving_plane;
    switch (small_airport.activity(current_time, moving_plane)){
      //  Let at most one Plane onto the Runway at current_time.
    case land:
      moving_plane.land(current_time);
      break;
    case takeoff:
      moving_plane.fly(current_time);
      break;
    case idle:
      run_idle(current_time);
    }
  }
  small_airport.shut_down(end_time);
}
```

在这个程序里，我们以对函数 initialize 的调用开始，这个函数为用户打印出指令，并且收集关于用户希望多长时间的模拟运行以及飞机场将有多忙的信息。然后我们进入一个 for 循环，在这个循环里，current_time 的取值范围是从 0 到用户描述的值 end_time。在每一个时间单元里，我们处理到达和起飞飞机的随机数字，这些飞机被声明并初始化成 current_plane 对象。在每一个循环中，我们也允许一架正在移动的飞机使用飞机跑道。如果没有飞机使用飞机跑道，就应用函数 run_idle。注意如果 Runway 类不能将一架即将到来的飞机加入到着陆 Queue 中(假设是因为队列满了)，那么就应用一个称为 refuse 的方法去指导 Plane 飞到另一个飞机场。类似地，有时不得不拒绝一架 Plane 起飞的许可。

3.5.3 随机数

模拟中的一个关键步骤是决定在每个时间单元内有多少架新飞机准备着陆或起飞。虽然可以用许多方法做出这种决定，但最有趣和最有用的一种方法是做出随机决定。当程序以随机决定的方式重复运行时，一遍运行与另一遍运行的结果将互不相同。通过充分的实验，模拟过程可能显示出与所研究的真实系统不相同的行为变化。Random 的方法 poisson 在前面的主程序里返回某个特定时间单元内正在准备着陆或起飞的飞机的随机数目。

* * * 伪随机数

附录 B 研究在计算机程序中使用的称为伪随机数(pseudorandom)的数字。几种不同种类的伪随机数对不同的应用是有用的。对于飞机场模拟，需要更高级的一种，称为泊松(Poisson)随机数。

为了介绍这个概念，先看一下下面的说法：平均每个家庭有 2.6 个孩子并不意味着每个家庭有 2 个孩子然后再加上 0.6 个孩子，而是意味着，通过对许多家庭的孩子数求平均，平均的孩子数是 2.6。因此，5 个家庭可能分别有 4，1，0，3，5 个孩子，平均数为 2.6。类似

地，如果即将到达的准备着陆的飞机数在一个时间单元里分别为 2,0,0,1,4,1,0,0,0,1，则在一个时间单元里，即将到达的飞机的平均数是 0.9 架。

* * * 期望值，泊松分布

我们现在以一个称为随机数的期望值 υ 的固定数开始，非负整数的一个序列满足期望值为 υ 的泊松分布(Poisson distribution)意味着在较长的子序列上，序列中的整数的平均值接近 υ。附录 B 描述了一个产生随机整数的C++类，这个类根据给定期望值的泊松分布产生随机整数，这正是我们在飞机场模拟中所需要的。

3.5.4 Runway 类的说明

* * * 规则

飞机跑道 Runway 类需要维护两个飞机队列，我们称其为 landing 和 takeoff，用于保存正在等待的飞机。让飞机在地面上等待比让它在空中等待要好些，因此这个小的飞机场仅当没有飞机等待着陆时才允许飞机起飞。因此，Runway 的方法 activity 控制着对 Runway 的使用，它首先为等待着陆的飞机 Queue 的队头服务，并且仅当着陆的 Queue 为空时才允许 Plane 起飞。

本模拟的一个目的是收集有关飞机场利用的可能的数据，因此自然会使用 Runway 类本身去保存统计量，如被处理的飞机数、花在等待上的平均时间和被拒绝服务的飞机数(如果有的话)。这些细节反映在下面 Runway 类定义的各种数据成员中。

* * * Runway 的定义

```
enum Runway_activity {idle, land, takeoff};
class Runway {
public:
  Runway(int limit);
  Error_code can_land(const Plane &current);
  Error_code can_depart(const Plane &current);
  Runway_activity activity(int time, Plane &moving);
  void shut_down(int time) const;
private:
  Extended_queue landing;
  Extended_queue takeoff;
  int queue_limit;
  int num_land_requests;          //  number of planes asking to land
  int num_takeoff_requests;       //  number of planes asking to take off
  int num_landings;               //  number of planes that have landed
  int num_takeoffs;               //  number of planes that have taken off
  int num_land_accepted;          //  number of planes queued to land
  int num_takeoff_accepted;       //  number of planes queued to take off
  int num_land_refused;           //  number of landing planes refused
  int num_takeoff_refused;        //  number of departing planes refused
  int land_wait;                  //  total time of planes waiting to land
  int takeoff_wait;               //  total time of planes waiting to take off
  int idle_time;                  //  total time runway is idle
};
```

注意 Runway 类在它的成员中有两个队列，它的实现反映了语句中的 has-a 关系，即一

个飞机场有一个着陆队列和一个起飞队列。

3.5.5 Plane类说明

Plane类需要维护特定的Plane对象的数据，这个数据必须包括航班号、到达机场系统的时间、到达或者离开的Plane的状态。既然不希望客户程序修改这些信息，就应把它放在私有数据成员里。当我们在主程序里声明了一个Plane对象时，我们希望在构造对象时即初始化这3条信息，为此我们需要带有这3个参数的Plane类构造函数。然而，在其他时间，我们希望在构造Plane对象时不用初始化这些信息，因为它们的值或者是无关的或者将由其他方式确定。所以对Plane类，我们实际上需要两个构造函数，其中一个带有3个参数，另一个不带参数。

* * * 函数的多种版本

C++语言正好提供了我们需要的特征：它允许使用同样的标识符去按需命名尽可能多的函数，甚至是在单个代码块中，只要这些函数里没有两个具有同样类型的参数列表。当函数被调用时，C++编译器通过查看实际参数的个数和它们的类型就能断定要使用函数的哪个版本，它只要确定哪种形式的参数集合在数目和类型上与实际参数匹配。

* * * 函数重载

当几个不同的函数使用单一的名称时，我们说这个名称被重载。就Plane类来说，我们能重载两个Plane构造函数，其中第一个函数使用了空的参数表，而第二个函数使用了3个整型变量的参数表。

从现在开始，类说明将经常包含两个构造函数，其中一个带参数的函数用于初始化数据成员，另一个则不带参数。

最后，Plane类必须包含方法refuse，land和fly，并供主程序显式地使用。我们也需要每个Plane能够与Runway类沟通到达机场的时间，因此最后一个称为started的方法将为此目的被包含进来。现在给出Plane类的说明。

* * * Plane的定义

```
enum Plane_status {null, arriving, departing};

class Plane {
public:
  Plane();
  Plane(int flt, int time, Plane_status status);
  void refuse()const;
  void land(int time)const;
  void fly(int time)const;
  int started()const;

private:
  int flt_num;
  int clock_start;
  Plane_status state;
};
```

3.5.6 模拟的函数和方法

用于模拟这些步骤的函数和方法的操作总的来说是直接明了的，因此我们继续依次编写每个函数和方法，只在需要明确时才加上必要的注释。

1. 模拟的初始化

```
void initialize (int &end_time, int &queue_limit,
                 double &arrival_rate, double &departure_rate)
/*Pre: The user specifies the number of time units in the simulation, the maximal queue
       sizes permitted, and the expected arrival and departure rates for the airport.
  Post: The program prints instructions and initializes the parameters end_time, queue
        _limit , arrival_rate, and departure_rate to the specified values.
  Uses: utility function user_says_yes */
{
  cout<<"This program simulates an airport with only one runway."<<endl
      <<"One plane can land or depart in each unit of time."<<endl;
  cout<<"Up to what number of planes can be waiting tc land"
      <<"or take off at any time?"<<flush;
  cin >>queue limit;
  cout<<"How many units of time will the simulation run?"<<flush;
  cin >>end time;
  bool acceptable;
  do {
    cout<<"Expected number of arrivals per unit time?"<<flush;
    cin >>arrival rate;
    cout<<"Expected number of departures_per unit time?"<<flush;
    cin >>departure rate;
    if(arrival_rate<0.0 || departure_rate<0.0)
      cerr<<"These rates must be nonnegative."<<endl;
    else
      acceptable=true;
    if(acceptable&&arrival_rate+departure_rate>1.0)
      cerr<<"Safety Warning: This airport will become saturated. "<<endl;
  } while (! acceptable);
}
```

2. Runway 的初始化

```
Runway::Runway(int limit)
/*Post: The Runway data members are initialized to record no prior Runway use and to
        record the limit on queue sizes. */
{
  queue_limit=limit;
  num_land_requests=num_takeoff_requests=0;
  num_landings=num_takeoffs=0;
  num_land_refused=num_takeoff_refused=0;
  num_land_accepted=num_takeoff_accepted=0;
  land_wait=takeoff_wait=idle_time=0;
}
```

3. 接受新的 Plane 到达 Runway 队列

```
Error_code Runway::can_land(const Plane &current)
/*Post: If possible, the Plane current is added to the landing Queue; otherwise, an
        Error_code of overflow is returned. The Runway statistics are updated.
  Uses:class Extended_queue. */
{
  Error_code result;
  if(landing.size()<queue limit)
    result=landing.append(current);
  else
    result=fail;
  num_land_requests++;
  if(result!=success)
    num_land_refused++;
  else
    num_land_accepted++;
  return result;
}

Error_code Runway::can_depart(const Plane &current)
/*Post: If possible, the Plane current is added to the takeoff Queue; otherwise, an
        Error_code of overflow is returned. The Runway statistics are updated.
  Uses:class Extended_queue. */
{
  Error_code result;
  if(takeoff.size()<queue_limit)
    result=takeoff.append(current);
  else
    result=fail;
  num_takeoff_requests++;
  if(result!=success)
    num_takeoff_refused++;
  else
    num_takeoff_accepted++;
  return result;
}
```

4. 处理对 Runway 的访问

```
Runway_activity Runway::activity(int time, Plane &moving)
/*Post: If the landing Queue has entries, its front Plane is copied to the parameter
        moving and a result land is returned. Otherwise, if the takeoff Queue has
        entries, its front Plane is copied to the parameter moving and a result
        takeoff is returned. Otherwise, idle is returned. Runway statistics are
        updated.
  Uses:class Extended_queue . */
{
  Runway_activity in_progress;
  if(!landing.empty()){
    landing.retrieve(moving);
    land_wait+=time-moving.started();
```

```
    num_landings++;
    in_progress=land;
    landing.serve();
  }
  else if (!takeoff.empty()){
    takeoff.retrieve(moving);
    takeoff_wait+=time-moving.started();
    num_takeoffs++;
    in_progress=takeoff;
    takeoff.serve();
  }
  else {
    idle_time++;
    in_progress=idle;
  }
  return in_progress;
}
```

5. Plane 的初始化

```
Plane::Plane(int flt, int time, Plane_status status)
/*Post: The Plane data members flt_num, clock_start, and state are set to the values of
        the parameters flt, time and status, respectively. */
{
  flt_num=flt;
  clock_start=time;
  state=status;
  cout<<"Plane number"<<flt<<"ready to";
  if(status==arriving)
    cout<<"land."<<endl;
  else
    cout<<"take off."<<endl;
}
Plane::Plane()
/*Post: The Plane data members flt_num, clock_start, state are set to illegal default
        values. */
{
  flt_num=-1;
  clock_start=-1;
  state=null;
}
```

*** 空初始化

第二个构造函数执行一个空初始化操作。在许多程序设计里，不必为类提供这样的一个构造函数。然而在C++中，如果我们曾声明带有构造函数的对象数组，那么这个对象必须有一个明确的默认构造函数，默认构造函数是一个没有参数(或者对所有的参数有指定的默认值)的构造函数。每个 Runway 对象包含飞机队列，并且其中的每一个队列都采用一个飞机数组来实现。因此在模拟中，我们确实需要空的初始化操作。

6. 拒绝一架 Plane

```
void Plane::refuse()const
/*Post: Processes a Plane wanting to use Runway, when the queue is full. */
{
  cout<<"Plane number"<<flt_num;
  if(state==arriving)
    cout<<"directed to another airport"<<endl;
  else
    cout<<"told to try to takeoff again later"<<endl;
}
```

7. 处理到达的 Plane

```
void Plane::land(int time)const
/*Post: Processes a Plane that is landing at the specified time. */
{
  int wait=time-clock_start;
  cout<<time<<":Plane number"<<flt_num<<"landed after"
      <<wait<<" time unit"<<((wait==1)?"":"s")
      <<"in the takeoff queue."<<endl;
}
```

在这个函数里，我们使用了三元运算符?:在需要的地方追加一个“s”并得到如“1 个时间单元”或“2 个时间单元”这样的输出。

8. 处理一架离开的 Plane

```
void Plane::fly(int time)const
/*Post: Process a Plane that is taking off at the specified time. */
{
  int wait=time-clock_start;
  cout<<time<<":Plane number "<<flt num<<" took off after"
      <<wait<<" time unit"<<((wait==1)?"":"s")
      <<"in the takeoff queue."<<endl;
}
```

9. 沟通 Plane 的到达数据

```
int Plane::started()const
/*Post: Return the time that the Plane entered the airport system. */
{
  return clock_start;
}
```

10. 标记空闲的时间单元

```
void run_idle(int time)
/*Post: The specified time is printed with a message that the runway is idle. */
```

```
{
  cout<<time<<":Runway is idle."<<endl;
}
```

11. 完成模拟

```
void Runway::shut down(int time)const
/*Post:Runway usage statistics are summarized and printed. */
{
  cout<<"Simulation has conluded after"<<time<<"time units."<<endl
      <<"Total number of planes processed "
      <<(num_land_requests+num_takeoff_requests)<<endl
      <<"Total number of planes asking to land"
      <<num_land_requests<<endl
      <<"Total number of planes asking to take off"
      <<num_takeoff_requests<<endl
      <<"Total number of planes accepted for landing"
      <<num_land_accepted<<endl
      <<"Total number of planes accepted for takeoff"
      <<num_takeoff_accepted<<endl
      <<"Total number of planes refused for landing"
      <<num_land_refused<<endl
      <<"Total number of planes refused for takeoff "
      <<num_takeoff_refused<<endl
      <<"Total number of planes that landed"
      <<num_landings<<endl
      <<"Total number of planes that took off"
      <<num_takeoffs<<endl
      <<Total number of Planes left in landing queue"
      <<landing.size()<<endl
      <<"Total number of planes left in takeoff queue"
      <<takeoff.size()<<endl;
  cout<<"Percentage of time runway idle"
      <<100.0*((float)idle_time)/((float)time)<<"% "<<endl;
  cout<<"Average wait in landing queue"
      <<((float)land_wait)/((float)num_landings)<<"time units";
  cout<<endl<<"Average wait in takeoff queue"
      <<((float)takeoff_wait)/((float)num_takeoffs)
      <<"time units"<<endl;
  cout<<"Average observed rate of planes wanting to land"
      <<((float)num_land_requests)/((float)time)
      <<"per time unit"<<endl;
  cout<<"Average observed rate of planes wanting to take off"
      <<((float)num_takeoff_requests)/((float)time)
      <<"per time unit"<<endl;
}
```

3.5.7 试验结果

我们以飞机场模拟的试验运行的部分输出作为本节的结束。有一些时段飞机跑道是空的，另有一些时段可能有一个队列完全是满的，使得飞机不能再入内。如果再一次运行该模

拟程序，可能会得到不同于这里给出的结果，但如果赋给程序的期望值是相同的，那么总结两次运行得出的数据将具有某些一致性。

这个程序模拟只有一个跑道的飞机场。

在每个时间单元只有一架飞机可以着陆或起飞。

在任意的时间等待着陆或起飞的飞机数的最大数目是多少？ 5

模拟将运行多少时间单元？ 1000

每个时间单元预期到达的飞机数是多少?. 48

每个时间单元预期离开的飞机数是多少?. 48

飞机 0 准备起飞。

0：飞机 1 着陆；在队列的 0 单元。

起飞队列里，飞机 0 在 0 时间单元后起飞。

飞机 1 准备起飞。

1：起飞队列里，飞机 1 在 0 时间单元后起飞。

飞机 2 准备起飞。

飞机 3 准备起飞。

2：起飞队列里，飞机 2 在 0 时间单元后起飞。

飞机 4 准备着陆。

飞机 5 准备起飞。

3：起飞队列里，飞机 4 在 0 时间单元后着陆。

飞机 6 准备着陆。

飞机 7 准备着陆。

飞机 8 准备起飞。

飞机 9 准备起飞。

4：起飞队列里，飞机 6 在 0 时间单元后着陆。

飞机 10 准备着陆。

飞机 11 准备起飞。

5：起飞队列里，飞机 7 在 1 时间单元后着陆。

飞机 12 准备着陆。

6：起飞队列里，飞机 10 在 1 时间单元后着陆。

7：起飞队列里，飞机 12 在 1 时间单元后着陆。

飞机 13 准备着陆。

飞机 14 准备起飞。

* * * 起飞队列满

飞机 14 被告之稍后再尝试起飞。

8：起飞队列里，飞机 13 在 0 时间单元后着陆。

9：起飞队列里，飞机 3 在 7 时间单元后起飞。

10：起飞队列里，飞机 5 在 7 时间单元后起飞。

11：起飞队列里，飞机 8 在 7 时间单元后起飞。

飞机 15 准备起飞。

12:起飞队列里，飞机 9 在 8 时间单元后起飞。
飞机 16 准备着陆。
飞机 17 准备着陆。
13:起飞队列里，飞机 16 在 0 时间单元后着陆。
飞机 18 准备着陆。
14:起飞队列里，飞机 17 在 1 时间单元后着陆。
15:起飞队列里，飞机 18 在 1 时间单元后着陆。
飞机 19 准备着陆。
飞机 20 准备起飞。
16:起飞队列里，飞机 19 在 0 时间单元后着陆。
17:起飞队列里，飞机 11 在 12 时间单元后起飞。
18:起飞队列里，飞机 15 在 6 时间单元后起飞。
19:起飞队列里，飞机 20 在 3 时间单元后起飞。
* * * 两个队列同时空
20:飞机跑道空闲。①
最后，在更多步骤的模拟后，我们得到了一个统计的概括。
* * * 概括
模拟在 1000 时间单元后终止。

总共处理的飞机数为	970
总共要求着陆的飞机数为	484
总共要求起飞的飞机数为	486
总共接受着陆的飞机数为	484
总共接受起飞的飞机数为	423
总共拒绝着陆的飞机数为	0
总共拒绝起飞的飞机数为	63
总共着陆的飞机数为	483
总共起飞的飞机数为	422
总共留在着陆队列里的飞机数为	1
总共留在起飞队列里的飞机数为	1
飞机跑道空闲的时间百分比	9.5%
着陆队列里的平均等待	0.36646 时间单元
起飞队列里的平均等待	4.63744 时间单元
希望着陆的飞机平均被观察率	0.484 时间单元
希望起飞的飞机平均被观察率	0.486 时间单元

注意最后的两个统计值给出了飞机请求着陆和离开许可的速率的观测值，它与运行开始时设置的期望值(在合理的范围内)相匹配；这个结果使我们有一定的信心相信附录 B 中的伪随机数算法确实模拟了合适的泊松分布。

① 译者注：步骤 3,4,5,6,7,8,13,14,15,16 中的起飞队列应为着陆队列。

程序设计项目 3.5

P1. 将所有用于飞机场模拟的函数和方法组合成一个完整的程序。用飞机场模拟程序做若干次试运行实验，调整准备着陆和起飞的飞机数的期望值，并找出在飞机不会被拒绝服务的条件下这些数字的尽可能大的近似值。如果队列的长度增加或减少，那么这些值将会有什么变化？

P2. 修改模拟程序，使飞机场有两条飞机跑道，其中一条总是用于着陆，另一条总是用于起飞。比较双跑道机场能服务的总飞机数和单条飞机跑道的飞机场的相应数字，前者是否是后者的两倍？

P3. 修改模拟程序，使飞机场有两条飞机跑道，其中一条总是用于着陆，另一条总是用于起飞。如果某个队列是空的，那么两条跑道都能用于其他的队列。如果着陆队列总是满的并且另一架飞机要到达着陆，那么将停止起飞，并将两条跑道都用于清理搁置的着陆飞机。

P4. 修改模拟程序，使飞机场有3条飞机跑道，其中各保留一条总用于着陆和起飞，第三条用于着陆，但在着陆队列为空的情况下，第三条亦可用于起飞。

P5. 修改最初的模拟程序(单条跑道)，使得当每架飞机到达着陆时，它将有(作为它的数据成员的)一个(随机产生的)油位，以剩余的时间单元度量。如果飞机没有足够的油位在队列中等待，则允许它立即着陆。因此着陆队列里的飞机可能需要再等待附加的单元，因此可能用完自身的燃料。作为着陆函数的一部分检查这一点，并且查明在飞机由于燃料用尽而开始坠毁前机场有多忙。

P6. 写一个占位程序来代替随机数函数，这个占位程序既能用于调试程序又允许用户正确地控制每一个时间单元内每个队列到达的飞机数。

启示和易犯的错误

1. 在选择实现前，确信已在抽象层充分说明所有的数据结构及其相关操作。
2. 在多种实现间进行选择，考虑数据结构上必须的操作。
3. 如果类 A 的每个对象都有类 B 的对象所有的特征，则将 A 作为 B 的派生类实现。
4. 当声明一个基类的成员时，考虑派生类的要求。
5. 通过使用公共继承实现类间的 is-a 关系。
6. 通过分层实现类间的 has-a 关系。
7. 使用泊松随机变量模拟随机事件的发生。

复习题

3.1

1. 定义队列这个术语，在队列上可以执行什么操作？
2. 在线性数组里如何实现循环数组？

3. 列出队列的三种不同的实现。

4. 解释类间的 has-a 和 is-a 关系之间的区别。

3.4

5. 定义模拟这个术语。

进阶参考书目

队列是许多数据结构书中包含的一个标准的主题，大多数现代的课本都采用将数据结构及其操作的特性同数据结构的实现相分离的观点，以两本书为例：Jim Welsh, John Elder, and David Bustard, *Sequential Program Structures*, Prentice-Hall International, London, 1984, 385 pages.

Daniel F. Stubbs and Neil W. Webre, *Data Structures with Abstract Data Types and Pascal*, Brooks/Cole Publishing Company, Monterey, Calif, 1985, 459 pages.

关于队列的许多主题，如附加信息、历史提示和数学分析的最好来源是 Knuth，卷 1(参考第 2 章)。

关于计算机模拟的一个初步概论出现在 Byte 10(1985 年 10 月)第 149～251 页，华盛顿国家飞机场的模拟出现在第 186～190 页。

下面这本书给出了关于类间可能的关系及这些关系恰当的C++实现的有用的讨论：Scott Meyers, *Effective C++*, second edition, Addison-Wesley, Reading, Mass., 1997.

第 4 章　链栈和链式队列

本章介绍数据结构的链式实现。我们从回顾C++动态内存分配的使用开始，然后介绍链栈和队列的实现。作为应用，我们派生了一个类用来表示多项式，并用它实现一个用于多项式的逆波兰计算器。本章以抽象数据类型的原理的复习作为结束。

4.1　指针和链式结构

4.1.1　引言和概论

1. 溢出问题

如果我们是通过将所有的数据存储在数组中的方式来实现数据结构，那么在编写程序时，就要将数组声明成某个固定的大小，而且在程序运行期间数组的大小也不能改变。为此，在编写程序时，我们必须决定数组需要的最大内存量并在声明时将它们预留出来。如果我们在较小的样本上运行这个程序，那么大多数的空间都不会被使用。而如果在较大的数据集合上运行这个程序，那么它又可能会耗尽所有预留的空间而遭遇溢出，甚至这时计算机内存并未全部使用，产生这种情况的原因仅仅是因为数组的初始边界太小。

＊＊＊空间的错误分配

即使我们很仔细地将数组定义得足够大，似乎可以用完所有可用的内存，我们仍会遇到溢出，因为虽然其他数组中仍有大量的未用空间，某个数组仍可能会达到它的限制。因为不同程序的不同运行可能导致不同的结构增长或缩小，所以不可能在程序实际执行前分辨出哪一个结构会溢出。

现代语言包括C++中都提供某些结构，允许我们不用数组就能将数据结构保存在内存中，借此可以避免溢出。

2. 指针

我们使用的C++结构是指针。指针也称为链或引用，它被定义为一个对象，经常是一个变量，用于存储其他对象的位置(也即机器的地址)，而这些对象典型地是一个结构，包含了我们要处理的数据。如果使用指针在所有数据中查找我们感兴趣的数据，那么我们不再需要关心数据本身实际保存的位置，在使用指针后，只要需要，我们就可以让计算机系统自己去查找数据。

3. 图表约定

图 4.1 显示了指向几种对象的指针。指针通常用箭头描述，而被引用的对象通常用矩形框表示。在本书所有的图中，通常将包含指针的变量显示成从浅灰色的框或圆圈中发起，浅灰色的圆圈一般表示包含指针的普通变量，浅灰色的框所包含的指针属于较大对象的一

部分。①

* * * 不指向任何位置的指针

因此，在图中，r 是一个指向对象“Lynn”的指针，v 是指向对象“Jack”的指针。我们能看到，指针的使用非常灵活：两个指针可以引用同一对象，比如图 4.1 中的 t 和 u，或者指针可以根本不引用任何对象。在图中我们将后一种情况表示为电气接地符号，如指针 s 所示。

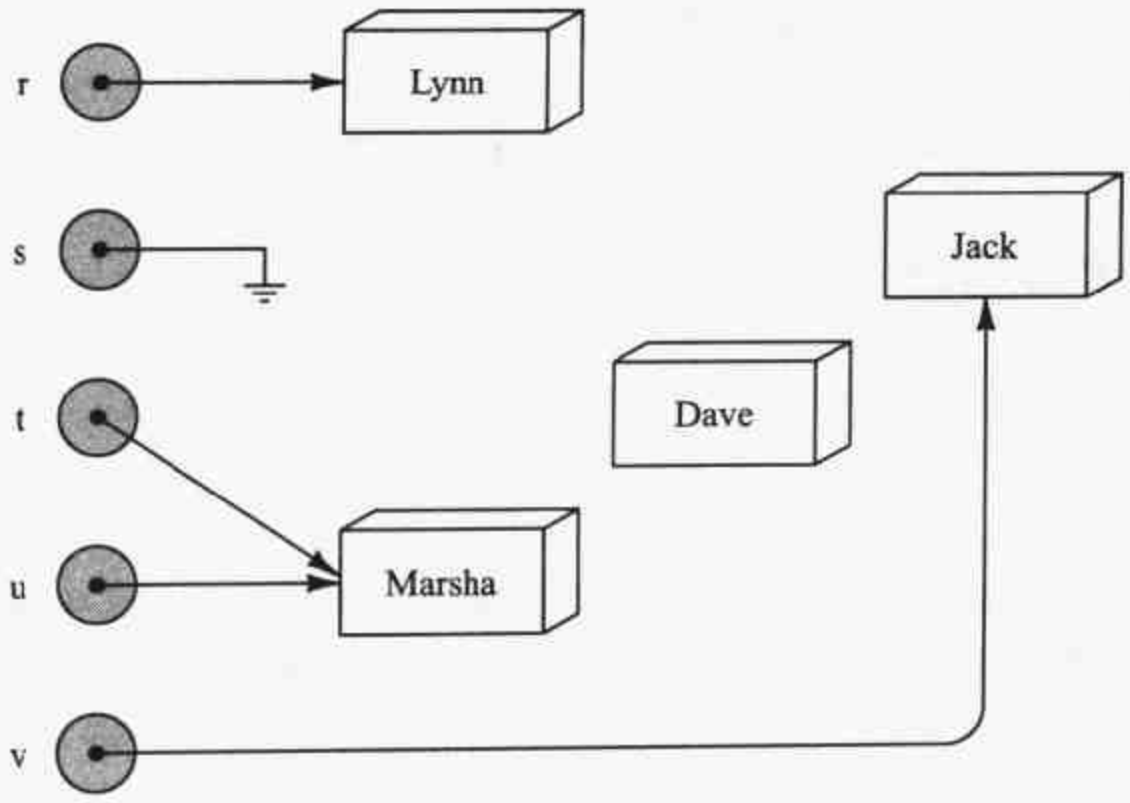


图 4.1 指向对象的指针

* * * 垃圾

使用指针时必须特别小心，而且要确信删除指针时没有对象丢失。在图中，对象“Dave”被丢失了，没有指针指向它，因此再也没有办法找到它。在这样的情况下，我们说对象已经变成了垃圾。虽然少量的垃圾不会造成危害，但如果任由垃圾堆积，那么它们最终将占据计算机中所有的可用内存并中断我们的程序。因此，我们在工作中总要努力避免垃圾的产生。

4. 链式结构

* * * 链表

在第 2 章和第 3 章，我们通过将关联结构的元素存储在数组中来实现栈和队列。在本章，我们将举例说明如何使用指针得到其不同的实现，在这种实现方式中，结构的元素被链到一起。链表(linked list)的思想是为表结构的每个元素扩充一个指针，这个指针给出了表中下一个元素的位置，图 4.2 说明了这种思想。

* * * 类比

从图中可以看出，链表在概念上比较简单，它使用了与小孩子玩的寻宝游戏同样的思想，其中发现的每个线索都指出在何处可以找到下一个线索。或者我们也可以考虑朋友间传递一盘流行磁带，Fred 有一盘流行磁带，已承诺给 Jackie。Carol 询问 Jackie 是否能借给

① 编者注：因本书非彩印，故原图中的着色部分(大部分为红色，还涉及相关图解文字等)处理为浅灰色。尤其在讲述红-黑树时，原图的红色对应于本中译版的浅灰色，敬请读者留意。

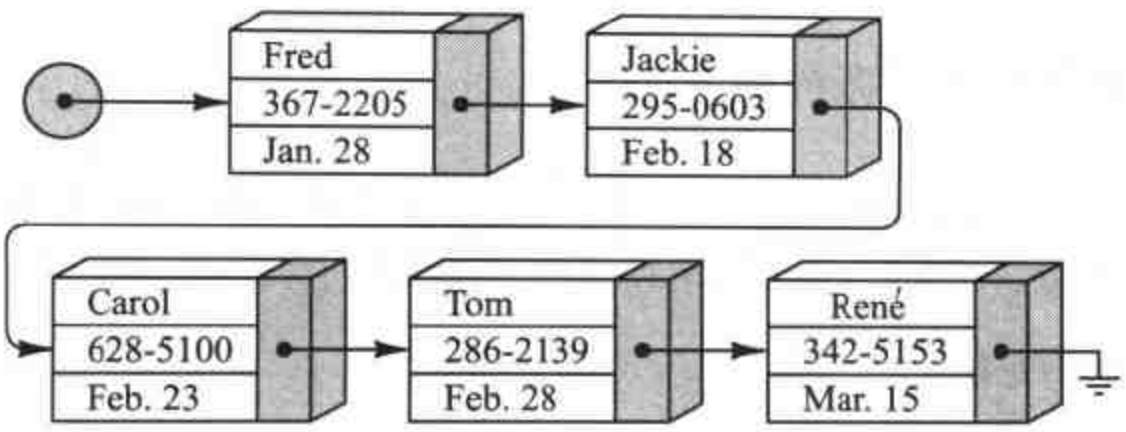


图 4.2　链表

他，然后又与 Tom 共享，如此下去。链表也可以被类比成命令的传递，仅仅在前一个任务完成的基础上才给出下一个命令。由于每个任务可以指定新的命令，而且无法预先得知有多少命令，因此所做的任务在数目上没有内在的限制。另一方面，2.2 节中学习的栈实现类似于写在单张纸上的命令列表，可能事先看见所有的命令，但写在单张纸上的命令数是有限的。

当积累了一些使用链表的实践经验后，就会发现链式结构就像在数组中实现的结构一样易于使用，然而方法却是截然不同的，因此我们必须花一些时间设计新的程序设计技术。在开始这个工作之前，我们先考虑一些更全面的观察。

5. 顺序表和链表

* * * 定义

顺序(contiguous)的意思是联系、接近和毗连。数组里的元素是顺序相接的，而且我们将保存在数组里的表称为顺序表。我们可以按要求给出顺序表和链表之间的区别，使用未受限制的词 list 只是为了同时包含两者。同样的约定亦适用于栈、队列和其他的数据结构。

6. 动态内存分配

* * * 多任务和分时

* * * 动态内存分配的优势

指针的使用不仅避免了由于用光数组的空间而引起不必要的溢出问题，在多任务或分时环境中也有它的优势。如果使用数组预先保留任务可能需要的最大数目的内存，那么这些内存在分配给它以后对其他的任务不再可用。如果必须将任务换出内存，那么当未用内存反复地拷入拷出磁盘时可能会有时间上的损失。我们能以一个非常小的内存空间开始，除了只保存程序指令和简单变量的空间外，无论何时有更多的数据需要空间时，能够向系统请求所需的内存，从而替代使用数组保存所有的数据。类似地，当某个项不再使用时，它所占有的空间能够返还给系统，系统然后可以将它分配给另一个任务。以这种方式，程序能从小的内存开始，仅当必要时才增长，因此当内存较小时，它能更有效地运行，而必要时其空间可以增长到计算机系统的界限。

即使每次只有一个任务运行，内存的这种动态控制方法仍证明是有用的。在任务的某一部分运行期间，某个目的可能需要大量的内存，这些内存随后被释放，并能重新分配给另一目的，也许这部分内存现在包含的是与以前完全不同类型的数据。

4.1.2 C++的指针和动态内存分配

包括C++在内的大多数现代程序设计语言都提供了强大的处理指针的机制，也有标准的函数用于在程序运行期间请求分配附加的内存和释放内存。

1. 自动对象和动态对象

*** 自动对象

*** 动态对象

C++程序执行期间能使用的对象有两种：自动对象和动态对象。自动对象是那些像平常一样在编写程序时声明和命名的对象，这些对象的空间由编译器显式分配，而且只要声明它们的程序块还在运行，它们就会一直存在。程序设计员不必担心用于自动对象的存储空间是否存在或者用于自动对象的存储空间是否会在使用后被清除。动态对象在执行期间被创建(也许被销毁)。由于动态对象在程序编译时并不存在，而仅是在运行时存在，所以在编写程序时并没有给它们命名。而且，被动态对象占用的存储必须由程序设计者完全管理。

获得动态对象的惟一方法是使用指针，然而，动态对象一旦被创建，它确实包含数据而且像其他对象一样具有类型。因此我们可以说创建 x 类型的一个新的动态对象并设置一个指针指向它，或者将指针从 x 类型的一个动态对象上移到另一个对象上，或者向系统返回一个 x 类型的动态对象。

另一方面，自动对象在声明它们的块执行期间不能显式地创建或销毁，它们在块开始执行时自动存在，当执行结束时自动消失。

*** 别名

指针变量可能用于指向自动对象，这样就产生了对象的又一个名称或别名。然后可以用其中的一个名称改变这个对象，随后再用另一个名称去使用它，我们在使用它的时候或许根本没有意识到它已被改变了。因此别名是危险的，应尽可能地予以避免。

2. C++符号

*** 指针类型

C++使用了一个星号(*)表示指针。如果 Item 表示我们感兴趣的数据类型，若一个指针指向这种类型的对象，那么此指针的类型为 Item *。例如我们声明：Item * item_ptr;。任何C++声明的最简单的语言翻译方式是从右至左读取声明的标记。在这个例子里，从右边开始读取的标记是 item_ptr，*，Item。因此，我们认为这个声明表示 item_ptr 是一个指向 Item 对象的指针。

我们通常仅使用 Item *类型的指针去存储 Item 类型的地址。然而，我们以后将看到，在 Item *类型的指针中存储来自 Item 的派生类的对象的地址也是合理的。

3. 创建和销毁动态对象

*** 动态对象的创建

可以用C++的运算符 new 创建动态对象。用一个对象类型作为运算符 new 的参数来

调用 new，它将返回新创建的具有恰当类型的动态对象的地址。如果要使用这个动态对象，就应立即将它的地址记录在一个合适的指针变量里：因此 new 运算符最经常用在指针赋值符号的右部。例如，假设 p 已经被声明为 Item 类型的指针，那么语句 p=**new** Item；将创建一个新的 Item 类型的动态对象并将它的位置赋值给指针 p。

＊＊＊自由存储区

我们创建的动态对象实际上保存在称为自由存储区(free store，或称为堆，即 heap)的计算机内存中。像所有的其他资源一样，自由存储区是有限的，如果创建了足够多的动态对象，则可能导致自由存储区被用尽。如果自由存储区是满的，那么对 new 运算符的调用将导致失败。在C++的早期实现中，这种失败将返回值 0 而非一个合法的机器地址。在 ANSI C++中，将产生一个异常表示 new 操作的失败。然而，修改后的语句 p=**new** (nothrow) Item；恢复了 new 运算符的传统的行为。在本课本中将采用 new 运算符的较老的行为。我们的代码将使用 nothrow 修改后运行在 ANSI C++环境中，它表示我们总是检查 new 的调用结果不是 0，从而确信调用成功。

＊＊＊动态对象的删除

为了帮助节省自由存储区，C++又提供了称为 delete 的运算符用于释放动态分配的对象，这个对象先前占用的存储空间将返还给自由存储区重新使用。运算符 delete 用于引用动态对象的指针，并且向系统返回被引用对象使用的空间。例如，如果 p 是指向一个 Item 类型的动态对象的指针，则语句 **delete** p；释放此对象。在这条 delete 语句执行后，指针变量 p 未定义，因此不能再被使用，除非它被赋予了一个新值。

new 和 delete 运算符的效果在图 4.3 中说明。

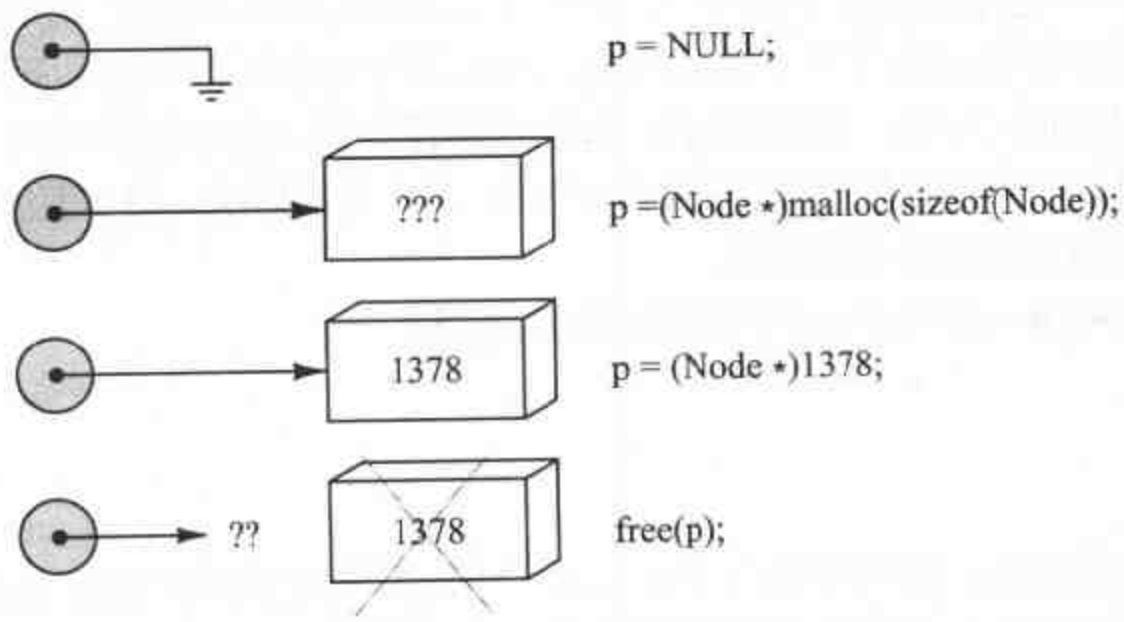


图 4.3　创建和释放动态对象

4. 跟随指针

＊＊＊解除引用指针

我们不仅在C++程序的声明中使用＊表示指针，而且也用＊来访问指针所引用的对象。此时星号不是出现在类型的右边，而是在指针的左边，因此＊p 表示 p 指向的对象。此外，链(link)和引用(reference)这两个词经常用于这种联系。取＊p 的操作有时候称为“解除引用指针 p。”

＊＊＊可修改的左值

注意被解除引用的指针，如＊p，实际上只是一个对象的名称。特别地，我们能在赋值号的左边使用表达式＊p。（从技术上说，被解除引用的指针是一个可修改的左值。）例如，赋值表达式＊p=0将被p引用的对象的值重置为0，此赋值在图4.4里予以说明。

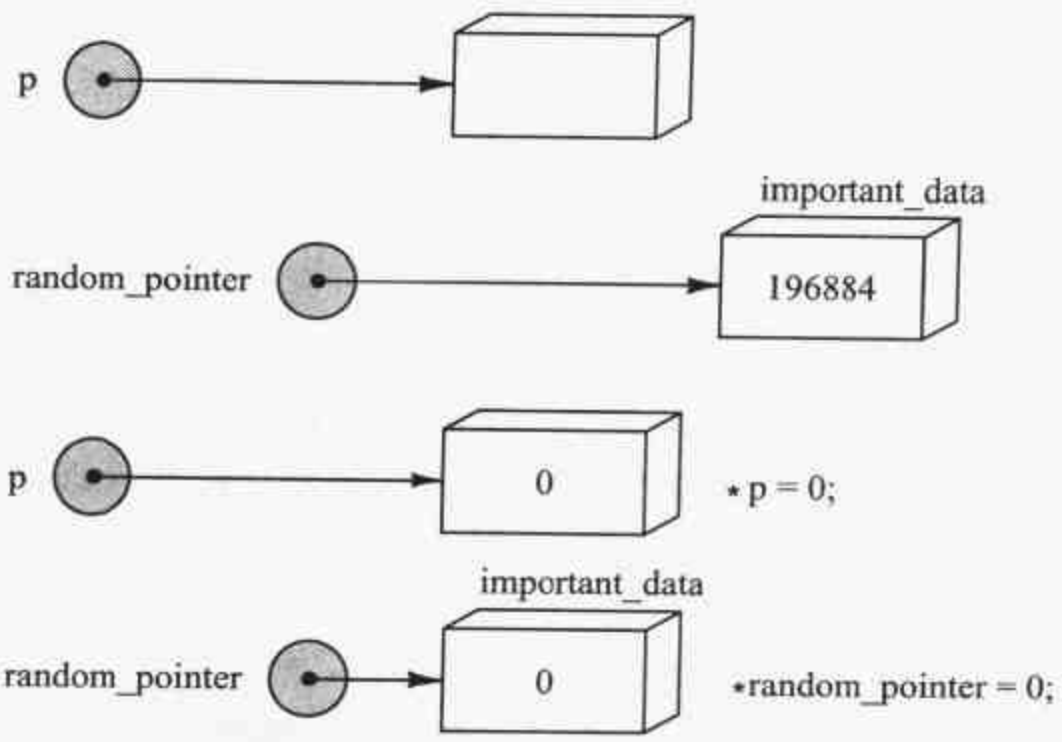


图4.4 修改被解除引用的指针

这个图也说明，如果指针random_pointer存储了非法的或未初始化的内存地址，则给对象＊random_pointer的赋值可能引起特别隐伏的错误。例如，未初始化的指针random_pointer可能恰好记录了另一个变量的地址，这个变量比如是important_data。任何像＊random_pointer=0这样的赋值将（意外地）改变important_data的值。如果幸运的话，这个错误将仅仅导致程序在以后的某个时间崩溃。然而因为这种崩溃可能不是立即发生的，所以它的源头将相当难以解释。更糟糕的是，我们的程序可能一直运行到完成，并且偷偷地产生不可靠的结果，或者它可能破坏内存的某个部分，其产生的影响一直到后来我们启动某个完全不相关的应用才可能会变得明显。

我们现在应考虑一种处理此类错误的有用的措施。

5. 空指针

＊＊＊空指针

有时一个指针变量p在当前没有引用动态对象，则可由下面的赋值运算建立空指针：p=NULL;，随后可以由如下的条件进行检验：**if**（p!=NULL）...。

在图中我们将电气接地符号保留给NULL指针使用。



值NULL作为常量对所有的指针类型都同样地适用，它是通用的，因为同样的值能赋给任何类型的指针变量。实际上，值NULL不是C++语言的一部分，但它在标准头文件如<cstddef>中被定义为0，此文件包含在我们的实用程序的头文件中。

＊＊＊未定义的指针与NULL指针的比较

仔细地观察未定义值的指针变量和值为NULL的指针变量之间的区别。断言p==

NULL 意味着 p 当前未指向任何动态变量。而如果 p 的值未定义，则 p 可能指向内存中的任何随机位置。

如果 p 被设为 NULL，则任何试图建立表达式 * p 的行为将导致程序立即崩溃。虽然处理错误是件不愉快的事，但这种崩溃比我们先前看到的由随机指针的赋值引起的问题，更易于理解和纠正。

程序设计技术规则

未初始化或随机的指针总是应该重新设置为 NULL。

指针对象在删除后应该重新设置为 NULL。

6. 动态分配数组

* * * 动态数组

new 和 delete 关键字能用于分配和删除用作数组的邻接动态存储块，例如，如果 array_size 表示整型值，则声明：item_array=**new** Item[array_size];，创建一个 Item 对象的动态数组，这个数组的元素的下标从 0 到 array_size - 1。可以用表达式访问一个典型的元素，如 item_array[i]。

例如，我们可以从用户处读入数组大小，并且用下列语句创建和使用一个恰当的数组。结果产生的赋值在图 4.5 中说明。

```
int size, *dynamic_array, i;
cout<<"Enter an array size:"<<flush;
cin>>size;
dynamic_array=new int [size];
for (i=0; i<size; i++)dynamic_array[i]=i;
```

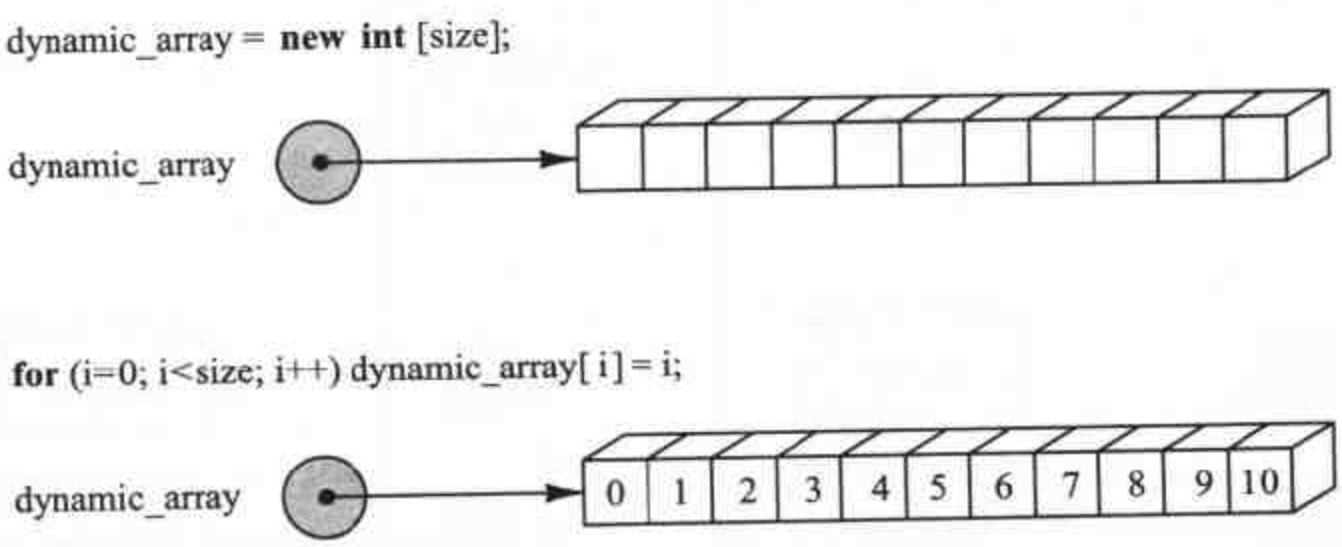


图 4.5　动态数组和指针

运算符 delete []用于返还动态分配的数组存储空间，例如用下面的语句返还 dynamic_array 中的存储空间：**delete**[] dynamic_array;。

7. 指针运算

* * * 指针运算

Item * 类型的一个指针对象 p 能参与赋值语句，能用于相等性检查，并且（作为一个参

数)能出现在函数调用里。程序设计员也能对指针值加、减一个整数,结果将得到另一个指针值。例如,如果 i 是一个整型值,则 p + i 是一个 Item * 类型的表达式。p + i 的值给出了从 p 开始,经过 i 个 Item 对象的内存地址偏移。也就是说,表达式 p + i 实际上产生地址 p + n * i,这里 n 是 Item 类型的简单对象所占用的存储空间的字节数。

输出指针的值也是可能的,不过它们是程序运行时赋予的地址,所以从程序的一次运行到下一次运行它们的值可能互不相同。而且它们的值(作为计算机内存里的地址)是程序设计员不应直接关心的实现特征。(然而,在调试阶段,有时输出指针的值是有用的,因为程序设计员能用它检查适当的等式成立以及进行了适当的指针赋值。)

注意,指针使用上的规则和限制不适用于此指针所引用的动态变量。如果 p 是一个指针,则 * p 通常不是一个指针(虽然指向指针的指针也是合法的),而是某种其他 Item 类型的变量,因此 * p 能用于 Item 类型的任何合法的方式中。

8. 指针赋值

* * * 赋值

关于赋值语句,记住 p=q 和 * p= * q 之间的区别是重要的,这两者都是合法的(假设 p 和 q 指向同样类型的对象),但二者有非常不同的效果。第一个语句使得 p 指向的对象还是 q 指向的对象,但它不改变那个对象的值或以前是 * p 的另一个对象的值,除非有某个其他的指针变量仍在引用它,否则后面的对象将变成垃圾。相反,第二个语句 * p= * q 将对象 * q 的值复制到对象 * p 中,因此现在我们有两个具有相同值的对象,p 和 q 分别指向两个单独的副本。最后,p= * q 和 * p=q 这两个赋值语句混合了类型并且是非法的(除了 p 和 q 都指向与它们具有同样类型的指针,而这种情况很少发生!)。图 4.6 说明了这些赋值。

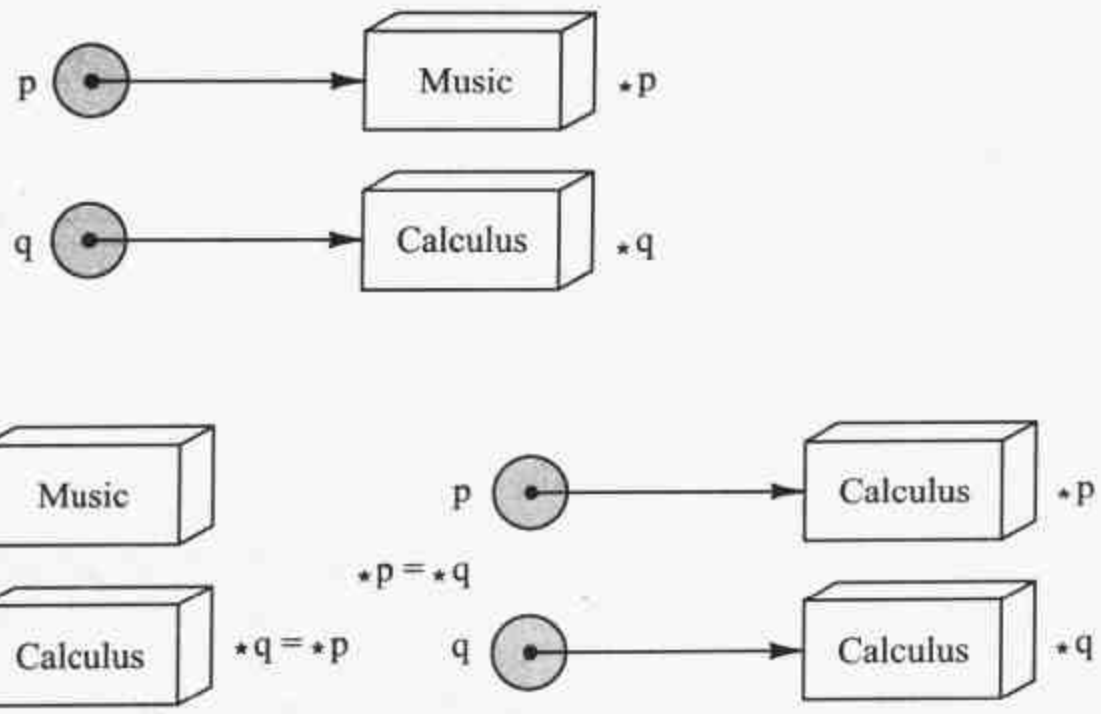


图 4.6 指针变量的赋值

9. 自动对象的地址

* * * 地址运算符

在C++里,自动对象通常只用它们的名称访问——就好像我们一直所做的那样,然而在

运行时，我们可以恢复和使用存储自动对象的机器地址。例如，如果 x 是类型为 Item 的自动对象，则 &x 是类型为 Item * 的值，这个值给出了 x 的地址。在这种情况下，像 Item * ptr=&x这样的声明和赋值将创建一个指向对象 x 的指针 ptr。

我们也可查看一下数组的起始元素存储的地址，可以通过使用未附上[]操作符的数组名找到这个地址。例如，给定声明 Item x[20]，赋值 Item * ptr=x 创建一个指针 ptr，它指向数组 x 的开始元素。注意赋值表达式 ptr=&(x[0])也能找到这个地址。按完全相同的方式，赋值表达式 p=&(x[i])定位于 x[i]存储的地址。然而，既然 x[i]的位置是 x[0]的位置偏移 i 项所要求的存储空间，因此表达式 x+i 使用指针运算给出了找到 x[i]的地址的简单方法。

10. 指向结构的指针

* * * 解除引用运算符->

许多程序使用了指向结构的指针，C++语言包含了一个额外的运算符帮助我们访问这类结构的成员。例如，如果 p 是指向一个对象的指针，并且这个对象有一个数据成员称为 the_data，那么我们能够用表达式(* p). the_data 访问这个数据成员。运算符的优先规则使我们不能省略该表达式中的圆括号，因此跟随链并查看一个成员的通用操作就变得麻烦起来。令人高兴的是，C++提供了简写的运算符->，使我们可以用一个具有同等功能但使用起来却更加方便的表达式 p->the_data 去代替表达式(* p). the_data。

比如，给定如下定义

```
class Fraction{
public:
  int numerator;
  int denominator;
};
Fraction *p;
```

我们能使用像 p->numerator=0 这样的表达式访问被 p 引用的 Fraction 对象的成员，或者也可以采用不太方便的但具有相同效果的表达式(* p). numerator=0 来访问。

4.1.3　链式结构的基础

具有了指针和指针类型这些工具，我们现在可以开始考虑C++里链式结构的实现问题，我们将从建立链式结构的项所需要的定义开始。

1. 结点和类型声明

* * * 结点和链

回忆图 4.2 中，链式结构由结点组成，每个结点同时包含了作为结构元素存储的信息和指示如何找到结构中下一个结点的指针。我们将把组成链式结构的这些结点看作结构的结点，其中的指针我们经常称之为链(links)。由于每个结点中的链都指示了从哪里找到结构的下一个结点，我们将采用名称 next 指明这个链。

我们将使用结构 struct 而不是类 class 来实现结点。struct 和 class 之间的惟一区别是，除非由关键字 private 和 public 修改，否则 struct 的成员是公共的而 class 的成员是私有的。

因此，通过使用修饰符 public 和 private 去调整成员的可见性，能够用 struct 或 class 实现任何结构。为此我们将采用约定：使用 struct 去实现未经封装的结构，它的所有成员都能被用户程序访问。例如，我们的结点结构就未经封装。然而，虽然能够凭它们本身的权限使用结点，但是我们通常应该将它们看作已封装的其他数据结构的 private 或 protected 的成员。由于这个原因，实现无封装的结点既安全又方便。这些决定转换成C++语言后即得到：

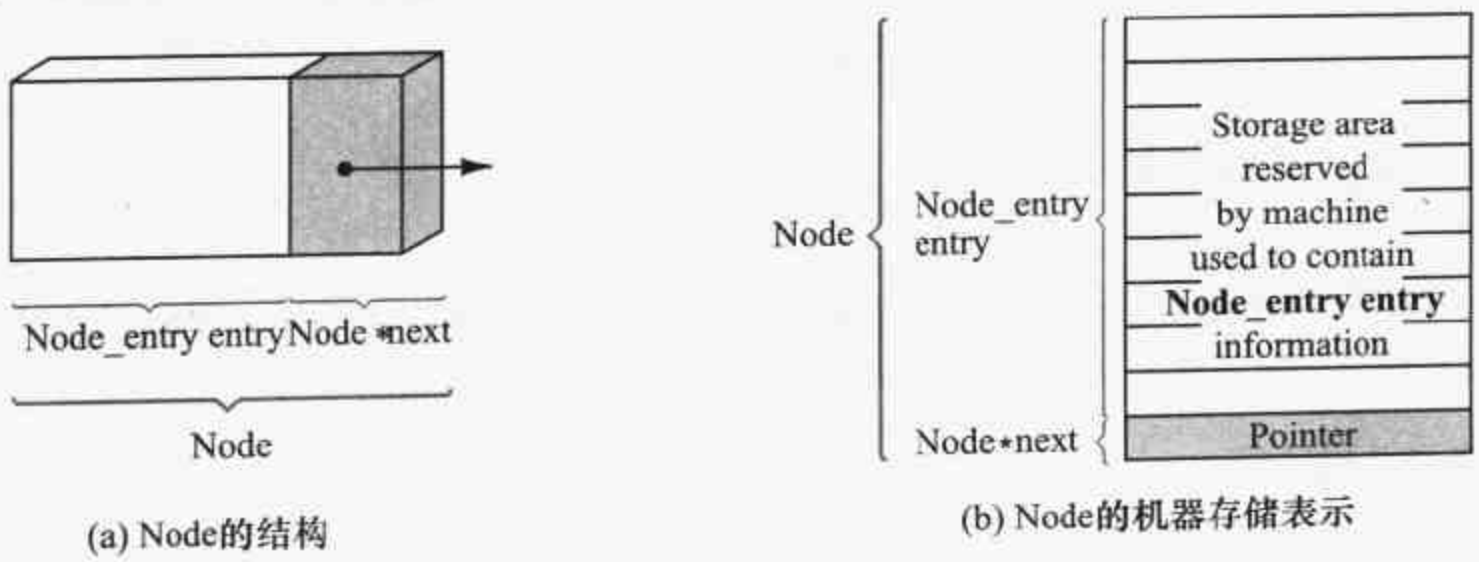
```
struct Node {
//  data members
  Node_entry entry;
  Node *next;
//  constructors
  Node();
  Node(Node_entry item, Node *add_on=NULL);
};
```

*** 在定义前使用

注意在这个定义里有一个明显的循环问题。Node * 类型的成员 next 是结构 Node 的一部分，而另一方面，看起来 Node 类型应该在 Node * 之前进行定义。为了避免这个循环定义的问题，对于指针类型(且只对指针类型)，C++放松了每个类型标识符必须在使用前予以定义这条基本规则。而即使 Some_type 还未定义，类型 Some_type * 在类型定义里也是合法的。(然而，Some_type 必须在程序结束前被定义，否则它将是一个错误。)

*** 指针所需的空间：存储字

为什么C++能这样放松它的规则而仍旧能有效地进行编译，其原因就在于不管指针指向什么类型的数据，所有的指针在内存中都占有同样大小的空间量，通常与一个整数所需的空间同样大。我们称指针占用的空间量为一个字(word)①，因此当遇到指针类型的声明时，编译器能为之预留出合适的存储空间，并会推迟检查所有的变量声明及使用与规则之间的一致性问题。例如，图 4.7 说明了实现 struct Node 定义所需要的结构和机器存储。



(a) Node的结构　　(b) Node的机器存储表示

图 4.7　包含指针的结构

① 更准确地说，大部分的计算机以 32 位存储一个整数，虽然 16 位整数和 64 位整数也较常见，而且一些机器还使用其他的长度，如 24 或 48 位。指针通常也占 32 位，但一些编译器使用其他的长度，如 16，24 或 64 位。一些编译器甚至使用两种指针长度，分别称为长指针和短指针。因此一个指针可能占用的空间通常称为半个字、一个字或双字，但我们采用的约定总是称指针的空间为一个字。

2. Node 的构造函数

＊＊＊重载

除了用于数据的存储外，结点说明中还包括两个构造函数，这些构造函数是以函数 Node::Node 的两种重载形式实现。在C++里，当两个或更多不同的函数实例被包含在程序中相同的作用域内部时，称这个函数被重载。当重载一个函数时，它的不同的实现应该有不同的集合或参数类型，从而编译器能够使用客户程序传递的参数去选择要使用的函数形式。例如，我们重载的构造函数的第一种形式具有空的参数列表，但第二种形式需要参数。

＊＊＊多重构造函数

第一个构造函数除了将 next 设为 NULL 作为错误检查的安全措施外不再做任何事情，第二个构造函数用于将 Node 的数据成员设置成参数指定的值。注意在第二个构造函数的原型中，为第二个参数指定了一个 NULL 的默认值，这就使得我们用两个参数或仅用第一个参数就能够调用第二个构造函数。在后一种情况中，第二个参数的默认值为 0。

```
Node::Node()
{
  next=NULL;
}
```

第二种形式接受两个参数用于初始化数据成员。

```
Node::Node(Node_entry item, Node*add_on)
{
  entry=item;
  next=add_on;
}
```

这些构造函数使得我们易于将结点连在一起形成链式构造。例如，下面的代码将产生图 4.8 中说明的链式结点。

```
Node first_node('a');                    //  Node first_node stores data 'a'.
Node*p0=&first_node;                     //  p0 points to first_Node .
Node*p1=new Node('b');                   //  A second node storing 'b' is created.
p0->next=p1;                             //  The second Node is linked after first_node.
Node*p2=new Node('c', p0);               //  A third Node storing 'c' is created.
  //  The third Node links back to the first node, *p0.
p1->next=p2;                             //  The third Node is linked after the second Node.
```

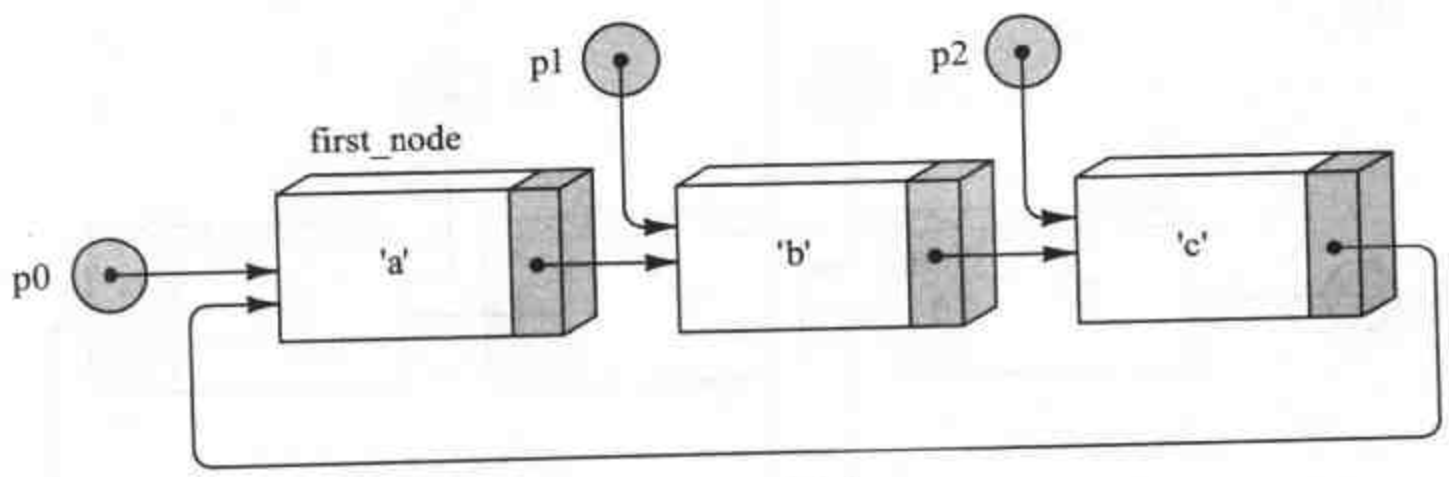


图 4.8　链式结点

注意，在图 4.8 和代码里，我们给第一个 Node 取名为 first_node，可以用这个名称访问

它，同时由于 p0 也指向它，因此也可用 * p0 访问它。然而，第二个和第三个结点没有明确的名称，因此最简单的方式是分别使用指针 p1 和 p2 访问它们。

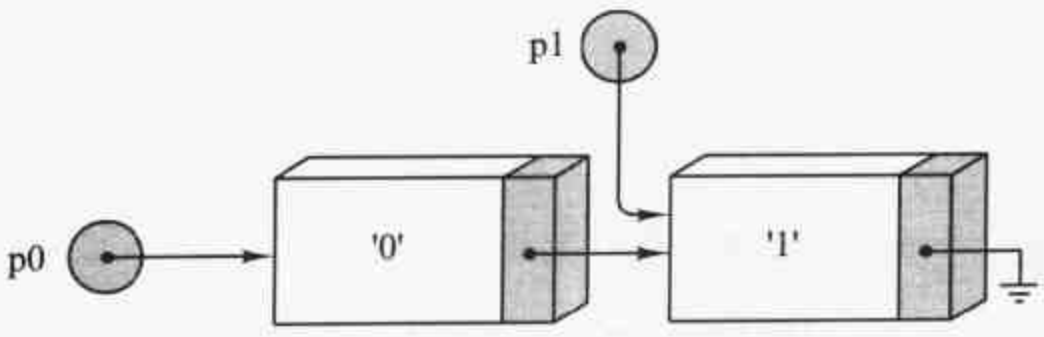
练习 4.1

E1. 画图说明由下列语句创建的链式结点的构造。

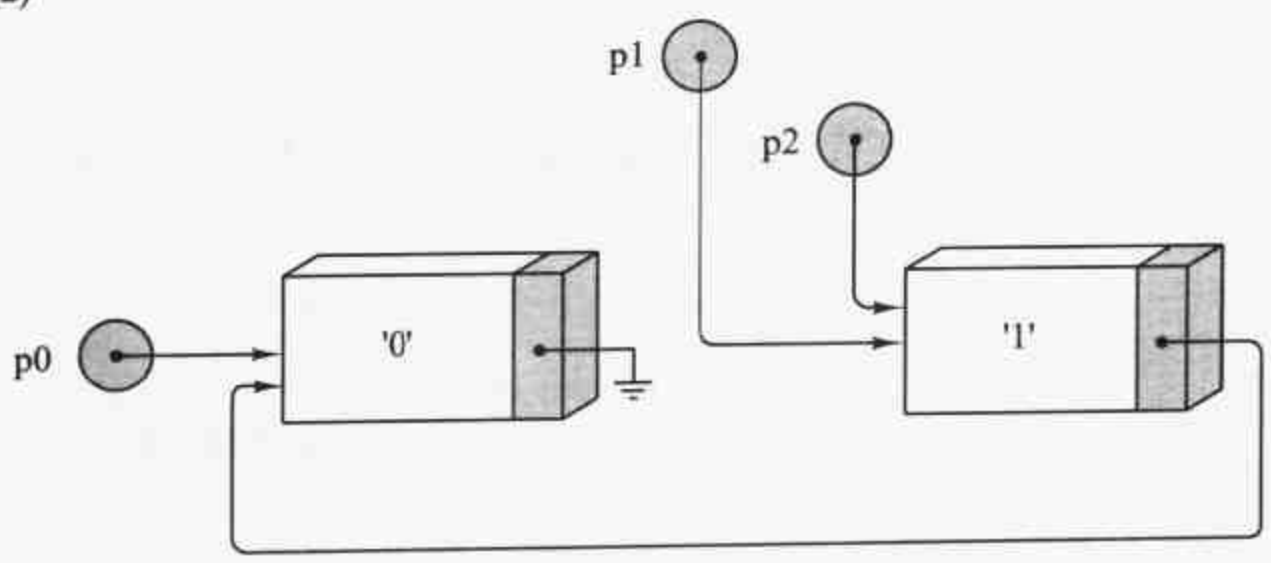
```
Node *p0=new Node('0');
Node *p1=p0->next=new Node('1');
Node *p2=p1->next=new Node('2', p1);
```

E2. 编写创建下列每个图中所示的结点的链式构造所需要的C++语句。对每一部分，将这些语句作为程序的一部分嵌入到打印每个结点内容(包括数据和下一指针 next)的程序中，从而论证结点已经正确地链接。

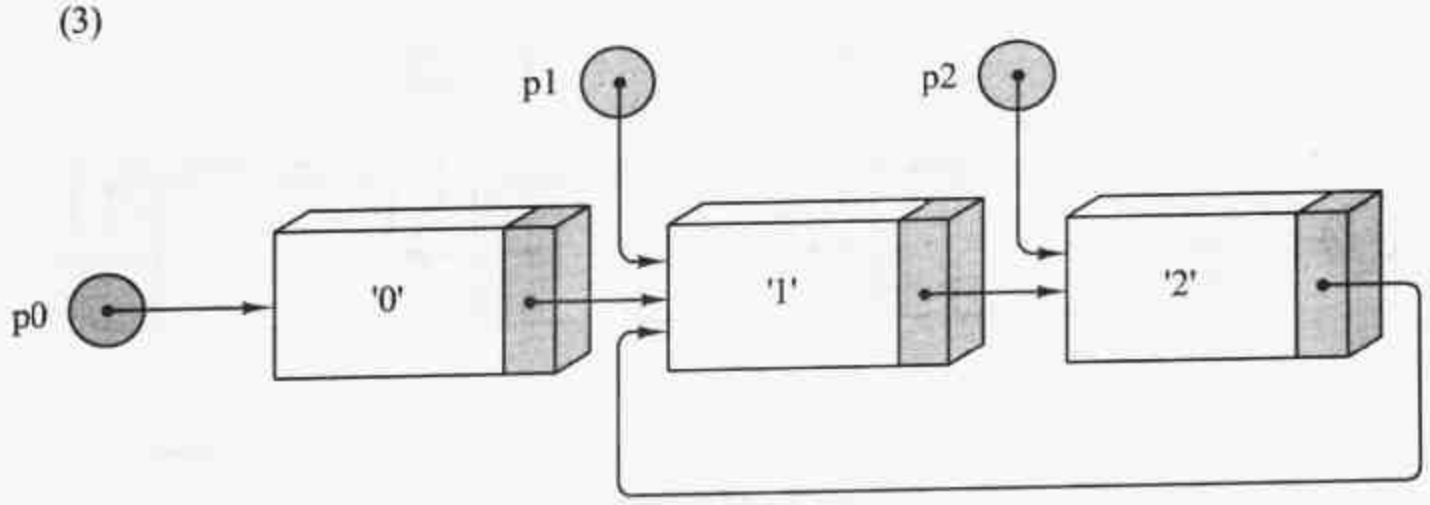
(1)



(2)



(3)



4.2　链栈

在 2.2 节中，我们使用元素数组去创建栈的顺序实现，在动态内存里以链式结构去实现栈同样简单。

* * * 一致性

我们是想让用户理解新的实现与先前的顺序实现是可换的，因此栈必须仍然包含Stack_entry 类型的元素。此外，我们打算用结点来构造栈，因此仍然需要声明 **typedef** Stack_entry Node_entry;，使存储在栈中的元素和结点具有相同的类型。而且，我们必须提供将 Stack_entry 类型的元素入栈和出栈的方法。在编写操作 push 和 pop 之前，我们必须考虑实际上这样的链栈是如何实现的等更多的细节。

现在要解决的第一个问题是确定究竟是采用链式结构的头部还是采用尾部作为栈顶。初看起来，好像(就顺序栈而论)在尾部增加结点可能更简单些，但是这种选择使得出栈变得困难：在链式结构里，由于存储在结构中的指针只给出了一个方向，因而没有快速的方法可以立即找出给定结点之前的结点。所以，在删除了最后的元素后，找到末尾的新元素可能还要从头开始一路追踪下来。为了方便链栈的出栈操作，将所有的增加和删除操作放在结构的头部做更好些。因此栈顶总是链式结构的首结点，如 4.9 图所示。

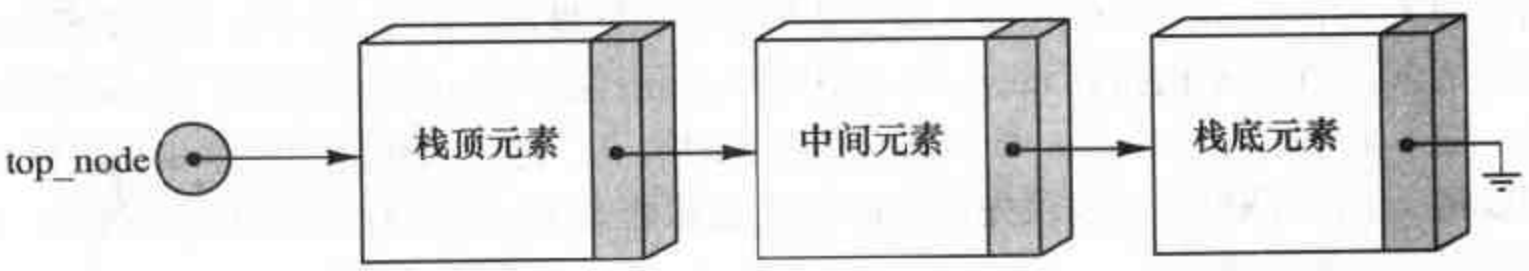


图 4.9　栈的链式形式

* * * Stack 类型的声明

每一个链式结构都需要有一个头成员指向它的首结点，对链栈来说，这个头成员总是指向栈顶。由于链式结构的每个结点都指向下一个结点，那么我们可以从头结点开始跟随指针到达链栈的所有结点。因此跟踪链栈里的数据所需要的惟一信息就是栈顶的位置。Stack 的底总可以看作是包含空(NULL)链的 Node。

因此我们可以通过建立类来定义链栈，这个类的惟一数据成员是栈顶。

```
class Stack {
public:
  Stack();
  bool empty() const;
  Error_code push(const Stack_entry &item);
  Error_code pop();
  Error_code top(Stack_entry &item) const;
protected:
  Node*top_node;
};
```

由于这个类仅包含一个数据成员，我们可能考虑免去这个类，并且用分配给栈本身的同样的名称去访问栈顶。然而，有 4 个原因表明要使用引入的类：

- 最重要的原因是维持封装：如果我们不使用类去包含栈，则将失去为栈建立方法的能力。
- 第二个原因是维持栈本身之间的逻辑区别，栈是由所有的元素（每一个在一个结点里）和栈顶组成，而栈顶是指向单一结点的指针。只需记录栈顶便可以找到栈中所有元素的事实与这种逻辑结构无关。
- 第三个原因是维持与其他数据结构和其他实现之间的一致性，在这些数据结构和实现中需要结构来集成若干方法和一些信息。
- 最后，让栈和指向栈顶的指针作为不相容的数据类型有助于编译器在调试时作出更好的类型检查。

＊＊＊空栈

我们从空栈开始，空栈意味着 top_node==NULL，现在考虑如何加入 Stack_entry item 作为第一个元素。首先我们必须在动态内存中创建一个新的 Node 用于存储 item 的副本，然后必须将存储在 new_top 中的地址复制到 Stack 的成员 top_node 中。因此，将 item 压入 Stack 栈的操作由下列命令组成：Node ＊new_top=new Node(item)；top_node=new_top；。注意创建 Node ＊new_top 的构造函数将它的 next 指针设为默认值 NULL。

＊＊＊链栈的入栈

往下继续时，我们假设已经有了一个非空的 Stack。为了将一个新的元素放入 Stack，需要向它加入一个 Stack_entry item。所需要的指针调整显示在图 4.10 中。首先必须创建一个新的结点，这个结点由指针 new_top 引用，new_top 保存 item 的值和指向 Stack 原栈顶的指针。接下来必须改变 top_node 以让它指向新的结点。这两次赋值的顺序相当重要：如果试图以相反的顺序赋值，则栈顶先前值的改变意味着 Stack 的原有部分将失去线索。我们因此得到下面的函数：

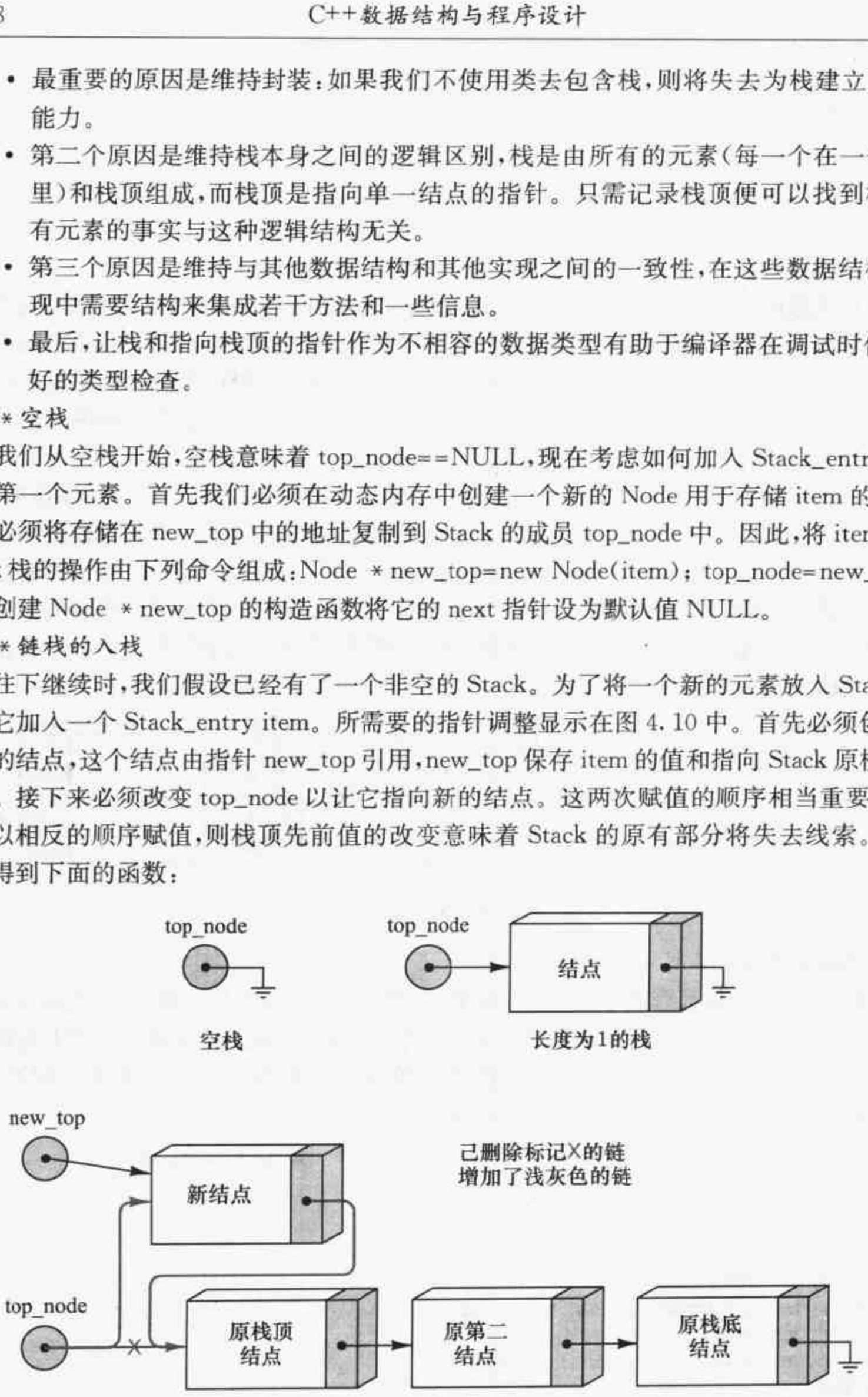


图 4.10 将一个结点压入链栈

```
Error_code Stack::push(const Stack_entry &item)
/*Post: Stack_entry item is added to the top of the Stack; returns success or returns a
```

```
        code of overflow if dynamic memory is exhausted. */
{
  Node *new_top=new Node(item, top_node);
  if (new_top==NULL)return overflow;
  top_node=new_top;
  return success;
}
```

当然，我们的基本操作必须符合先前的说明，因此仍要包含错误检查和考虑特殊情况。特别是，当不可能为 new_top 分配动态内存时应返回一个 Error_code。

此函数的一个极端情况是空的 Stack，即 top_node==NULL 的情况。注意，在此情况中，将第一个元素压入空的 Stack 和压入一个元素到非空的 Stack，函数的工作是一样的。

* * * 链栈的出栈

将一个元素从链栈出栈同样简单，这个过程在图 4.11 中加以说明，它的步骤可以对应到如下的C++代码。

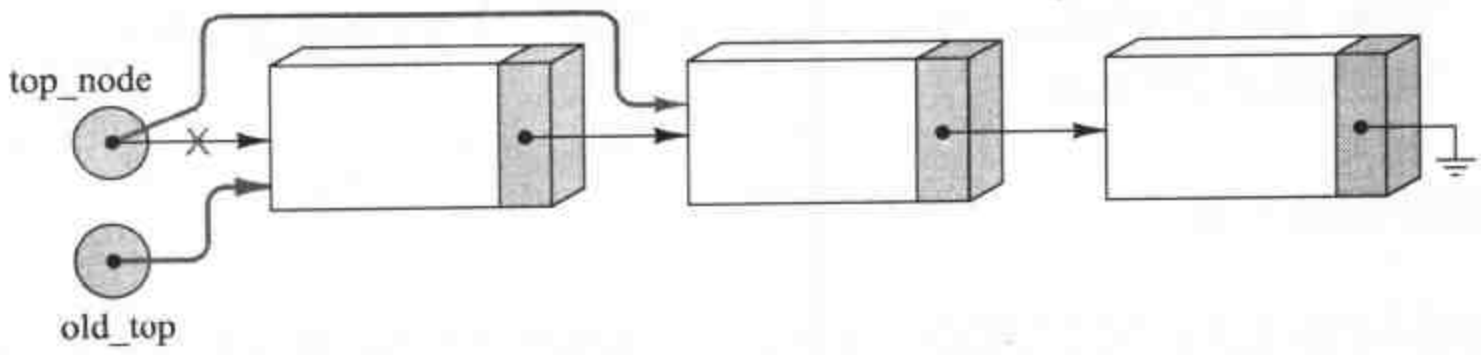


图 4.11　从链栈弹出一个结点

```
Error_code Stack::pop()
/* Post: The top of the  Stack is removed. If the Stack is empty the method returns
        underflow; otherwise it returns success . */
{
  Node *old_top=top_node;
  if (top_node==NULL)return underflow;
  top_node=old_top->next;
  delete old_top;
  return success;
}
```

当我们在方法 pop 里重置 top_node 的值时，指针 old_top 是到 Stack 栈顶位置上结点的惟一的链。这样，一旦函数结束，old_top 将失去余地，也没有办法再访问那个 Node，因此为了避免产生垃圾，需要删除 old_node。当然，在较小的应用中，如果不使用 delete，pop 方法同样工作得很好。但是如果用户程序重复使用这样的实现，那么垃圾最终将堆积得占据所有可能的内存，从而将妨碍用户程序的运行。

我们的链式实现实际上面临着许多微妙的缺陷，这一点将在下一节加以指明并予以纠正。我们急于表明自己所提供的方法没有错误，但用户程序却可能使 Stack 对象出错。为此必须在文档中说明 Stack 类的使用限制，或者必须在类中加进许多额外的特性以改正问题。

练习 4.2

E1. 解释为何不能将下面的实现用于我们的链式 Stack 中的 push 方法。

```
Error_code Stack::push(Stack_entry item)
{
  Node new_top(item, top_node);
  top_node=new_top;
  return success;
}
```

E2. 考虑包含方法 size 的链栈。由于栈里的元素个数不作为栈记录的独立成员保存，因此方法 size 需要对整个栈使用一个循环以计算元素的个数。

(1)编写链栈的方法 size，这个方法使用一个循环，依次移动指针去遍历栈。

(2)考虑修改链栈的声明，使得栈变成有两个成员的结构，包括栈顶和给出栈大小的计数器。那么链栈的其他方法需要做出什么修改？讨论这种修改与链栈的最初实现相比所具有的优缺点。

程序设计项目 4.2

P1. 编写一个演示程序，用于检验这一节里编写的操作栈的方法。模仿 3.4 节开发的程序并尽可能地使用它的代码，栈中的元素应为字符型。程序中要编写一个在线菜单供用户选择任何栈操作。程序在完成所要求的操作后，将告知用户结果并询问下一个需求。当用户希望将一个字符入栈时，程序需要询问把什么字符入栈。

使用栈的链式实现，并注意维护信息隐藏原理。

4.3 带保护的链栈

* * *析构函数、拷贝构造函数、重载的赋值运算符

当用户程序代码中应用上一节设计的链栈的方法时，可能导致垃圾的积累或打破 Stack 对象的封装。在这一节，我们将详细地检查这些不安全性是如何产生的，并且将查看C++语言提供的三种特殊的策略去缓和这些问题。这些策略采用附加的类方法的形式，它们是析构函数、拷贝构造函数和重载赋值运算符。这些新的方法代替编译器产生默认的行为并经常被称为是隐式的(即用户程序没有显式的动作)。因此对客户程序来说，此安全特征的加入并没有改变 Stack 类的面貌。

4.3.1 析构函数

假设客户程序运行一个简单的循环，这个循环中定义了一个 Stack 对象并有一些数据入栈。例如考虑如下代码：

```
for (int i=0; i<1000000; i++){
```

```
  Stack small;
  small.push(some_data);
}
```

* * * 垃圾的累积

每次循环都会创建一个 Stack 对象，并且将数据插入动态分配的内存，然后对象就超出了管理范围。现在假设客户程序使用 4.2 节的链式 Stack 实现，一旦对象 small 超出了范围，存储在 small 里的数据就变成了垃圾，而无数次循环过程将累积许多垃圾。我们不能因为这种问题去谴责客户程序的(明显罕见的)行为：此循环用顺序 Stack 实现在执行时不会有任何问题，采用这种方式，每次在 Stack 对象超出范围后，为其分配的所有数据空间都将被释放。

链栈实现的工作当然还包括编写文档警告客户程序不要让非空的 Stack 对象超出范围，或者在它们超出范围之前就清除 Stack 对象。

* * * 析构函数

C++语言提供称为析构函数的类方法用于解决这个问题。对每个类来说，析构函数是一个特殊的方法，它在类的对象超出范围前立即执行。而且，客户程序不必显式地调用析构函数，甚至不必知道它的存在。因此从客户程序的观点，带析构函数的类能简单地被不带析构函数的类替换。

析构函数经常用于删除动态分配的对象，否则这些对象将成为垃圾。在我们的例子里，将简单地为链式 Stack 类加进一个这样的析构函数。在此修改后，这个类的客户程序将不会让非空的 Stack 超出范围而产生垃圾。

* * * 析构函数的前缀~

析构函数必须声明为不带返回类型和参数的类方法，通过在相应的类名前面加一个前缀~来给出它的名称。因此，Stack 的析构函数的原型是：Stack::~Stack();。

由于我们所设计的 pop 方法能删除单个的结点，因此可以通过重复地弹出 Stack 的元素来实现 Stack 的析构函数。

```
Stack::~Stack()                    //  Destructor
/*Post: The Stack is cleared. */
{
  while (! empty())
    pop();
}
```

我们的策略是：每个链式结构都应配备一个析构函数，以在对象超出范围前清除它们。

4.3.2 重载赋值运算符

即使在我们为链栈的实现增加了析构函数以后，反常的用户程序仍然能用简单的循环产生巨大的垃圾堆积。例如，下面的客户程序代码首先会创建一个外部的 Stack 对象，然后运行一个循环，用命令建立并立即重置内部的 Stack。

```
Stack outer_stack;
for (int i=0; i<1000000; i ++){
```

```
  Stack inner_stack;
  inner_stack.push(some_data);
  inner_stack=outer_stack;
}
```

* * * 废弃的内存

语句 inner_stack=outer_stack 给 Stack 的实现造成一个严重的问题。C++通过复制数据成员 outer_stack. top_node 完成结果的赋值，此项复制重写了指针 inner_stack. top_node，因此 inner_stack 的内容丢失了。就像我们在图 4.12 里说明的，在每次循环时，先前的 inner_stack 的数据都变成了垃圾。结果产生的垃圾堆积当然应归罪于 Stack 的实现。同从前一样，当客户程序使用顺序栈实现时，不会出现任何问题。

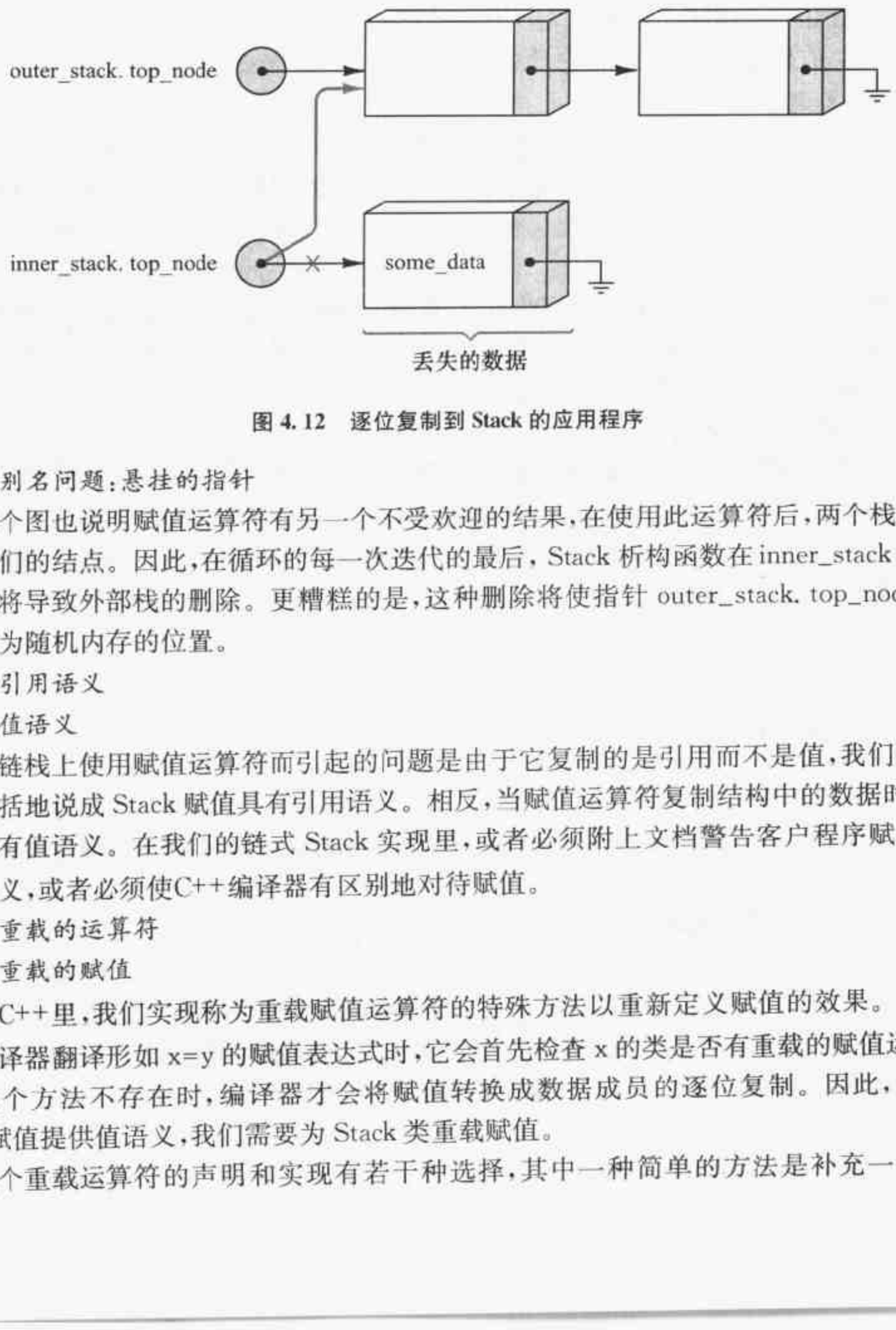


图 4.12 逐位复制到 Stack 的应用程序

* * * 别名问题：悬挂的指针

这个图也说明赋值运算符有另一个不受欢迎的结果，在使用此运算符后，两个栈对象将共享它们的结点。因此，在循环的每一次迭代的最后，Stack 析构函数在 inner_stack 上的任何运用将导致外部栈的删除。更糟糕的是，这种删除将使指针 outer_stack. top_node 寻址于已成为随机内存的位置。

* * * 引用语义

* * * 值语义

在链栈上使用赋值运算符而引起的问题是由于它复制的是引用而不是值，我们把这种情况概括地说成 Stack 赋值具有引用语义。相反，当赋值运算符复制结构中的数据时，我们说它具有值语义。在我们的链式 Stack 实现里，或者必须附上文档警告客户程序赋值具有引用语义，或者必须使C++编译器有区别地对待赋值。

* * * 重载的运算符

* * * 重载的赋值

在C++里，我们实现称为重载赋值运算符的特殊方法以重新定义赋值的效果。每次当C++编译器翻译形如 x=y 的赋值表达式时，它会首先检查 x 的类是否有重载的赋值运算符。仅当这个方法不存在时，编译器才会将赋值转换成数据成员的逐位复制。因此，为了给 Stack 赋值提供值语义，我们需要为 Stack 类重载赋值。

这个重载运算符的声明和实现有若干种选择，其中一种简单的方法是补充一个返回

void 类型的原型：**void** Stack ::operator =(**const** Stack & original)；

它声明了一个称为 operator=的 Stack 方法，operator= 是重载的赋值运算符，通常能够采用成员选择运算符来调用这个方法：x. **operator**=(y)；。作为选择，可以采用更自然、更方便的运算符语法调用这个方法：x=y；。

* * *运算符语法

通过查看操作数的类型，C++编译器能分辨出要使用的是重载运算符而不是普通的赋值。通过省略表示成员选择的句点、关键字 operator 和普通方法调用中的圆括号，我们可以得到运算符语法。

Stack 类的重载赋值运算符的实现证明相当复杂。

- 首先，必须对调用参数中排列的数据做一个副本。
- 其次，必须清除掉待赋值的 Stack 对象中的任何数据。
- 最后，必须将新近复制的数据移到 Stack 对象中。

```
void Stack:: operator = (const Stack &original)   // Overload assignment
/*Post: The Stack is reset as a copy of Stack original. */
{
  Node*new_top,*new_copy,*original_node=original.top_node;
  if (original_node==NULL)newtop=NULL;
  else {                        // Duplicate the linked nodes
    new_copy=new_top=new Node(original_node->entry);
    while (original_node->next!=NULL){
      original_node=original_node->next;
      new_copy->next=new Node(original_node->entry);
      new_copy=new_copy->next;
    }
  }
  while (!empty())              // Clean out old Stack entries
    pop();
  top_node=new_top;             // and replace them with new entries
}
```

注意，在实现中需要从赋值的那些 Stack 对象中弹出所有现存的元素。作为一种预防措施，首先要对 Stack 参数做一个副本，然后再重复运用方法 pop，这样可以确保赋值运算符在像 x=x 这样的赋值中没有丢失对象。

* * *遗留的缺陷：多重赋值

虽然重载赋值运算符成功地给出了 Stack 的赋值语义，但它还有一个缺陷：客户程序不能使用象 first_stack=second_stack=third_stack 这样的表达式中的赋值结果。一种非常彻底的实现将返回一个 Stack& 类型的引用以允许客户程序写出这样的表达式。

4.3.3　拷贝构造函数

链式结构可能发生的最后一种不安全性出现于C++编译器调用对象的副本时。例如，当一个参数通过值传递给一个函数时，需要复制对象。在C++中，默认的复制操作复制类中每个数据成员。如图 4.12 里说明的一样，在链式 Stack 上默认的复制操作导致对象间的数据共享。换言之，链式 Stack 上默认的复制操作具有引用语义，这样就允许怀有恶意的客户

声明和运行旨在破坏链式 Stack 对象的函数：

```
void destroy_the_stack (Stack copy)
{
}
int main()
{
  Stack vital_data;
  destroy_the_stack(vital_data);
}
```

在此代码中，Stack vital_data 的一个副本被传递给函数。Stack copy 与 Stack vital_data 共享其结点，因此在函数的末尾对 copy 运用 Stack 的析构函数时，vital_data 也受到了破坏。

* * * 拷贝构造函数

此外，C++提供了一个工具用于修正这个特殊的问题。实际上，如果包含拷贝构造函数作为 Stack 类的一个成员，则只要当编译器需要复制 Stack 对象时，就会调用我们的拷贝构造函数，因此能够保证 Stack 对象使用值语义复制。

对任意的类，声明拷贝构造函数的标准方法是带一个变量的构造函数，这个变量被声明为类的对象上的常量引用。因此，Stack 的拷贝构造函数通常具有以下原型：Stack::Stack (**const** Stack & original);。在这个构造函数的实现中，首先处理复制空 Stack 的情况，然后复制第一个结点，之后运行一个循环复制所有的其他结点。

```
Stack::Stack(const Stack &original)   //  copy constructor
/*Post: The Stack is initialized as a copy of Stack original. */
{
  Node*new_copy, *original_node=original.top_node;
  if (original_node==NULL)top_node=NULL;
  else {                        //  Duplicate the linked nodes.
    top_node=new_copy=new Node(original_node->entry);
    while (original_node->next!=NULL){
      original_node=original_node->next;
      new_copy->next=new Node(original_node->entry);
      new_copy=new_copy->next;
    }
  }
}
```

此代码类似于我们的重载赋值运算符的实现。然而在此情况中，由于我们正在创建一个新的 Stack 对象，因此就不必删除任何已有的栈元素。

总的来说，对于每个链式的类，或者应该包含拷贝构造函数，或者应该提供文档来警告客户程序对象以引用语义进行复制。

4.3.4 修改的链栈说明

我们通过更新链栈的说明来结束本节，这个说明里包含了所有提议的安全特性。

```
class Stack {
```

```
public:
//  Standard Stack methods
  Stack();
  bool empty() const;
  Error_code push(const Stack_entry &item);
  Error_code pop();
  Error_code top(Stack_entry &item) const;
//  Safety features for linked structures
  ~Stack();
  Stack(const Stack &original);
  void operator = (const Stack &original);
protected:
  Node *top_node;
};
```

练习 4.3

E1. 假设 x,y 和 z 是 Stack 对象。解释为何 4.3.2 节的重载赋值运算符不能用在像 x=y=z 这样的表达式中。修改这个重载的赋值运算符的原型和实现,使得此表达式变成合法的。

E2. 下面的代码试图使用拷贝构造函数实现链栈的重载赋值运算符,其中哪里有错?

```
void Stack::operator = (const Stack &original)
{
  Stack new_copy(original);
  top_node=new_copy.top_node;
}
```

如何修改这些代码以得到正确的实现?

4.4　链式队列

在邻式存储里,队列显然比栈更难以操作,因为它必须把线性存储当作好像是环形进行处理,并且空队列和满队列等极端情况也引起了困难。然而链式存储正是因为队列才真正

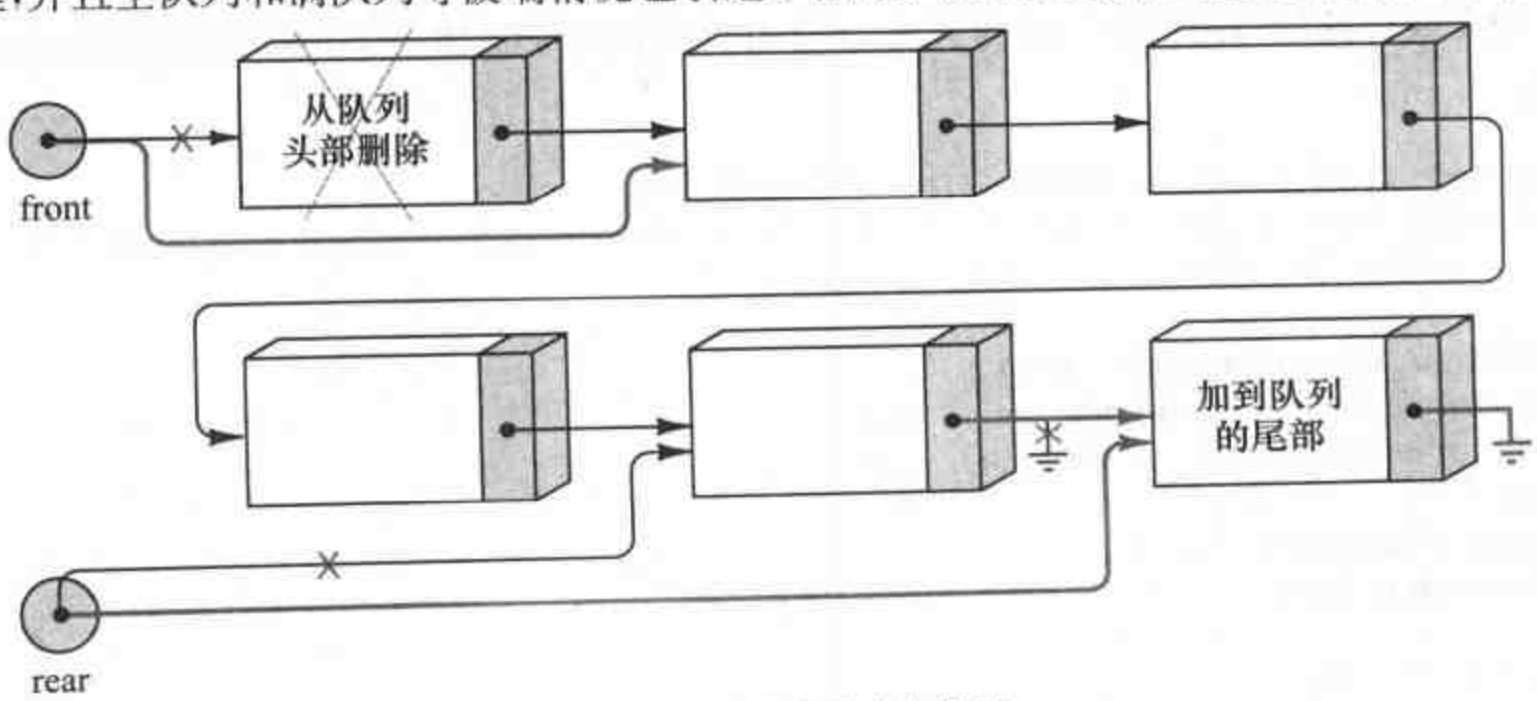


图 4.13　链式队列上的操作

盛行起来。链式队列像链栈一样易于处理，我们只需要保留两个指针：front 和 rear，它们分别指向队列的头部和尾部。图 4.13 中说明了插入和删除操作。

4.4.1 基本的定义

* * * 类型 Queue

对所有的队列，我们用 Queue_entry 表示队列里项的类型。对于链式实现，我们像 4.1.3 节里对链式结构所做的那样去定义结点，并且使用 typedef 语句标识类型 Queue_entry 和 Node_entry。与我们对栈所做的工作非常类似，我们得到下列说明：

```
class Queue {
public:
// standard Queue methods
  Queue();
  bool empty() const;
  Error_code append(const Queue_entry &item);
  Error_code serve( );
  Error_code retrieve(Queue_entry &item) const;
// safety features for linked structures
  ~Queue();
  Queue(const Queue &original);
  void operator = (const Queue &original);
protected:
  Node *front, *rear;
};
```

* * * 初始化

第一个构造函数将队列初始化为空，如下所示：

```
Queue::Queue()
/*Post: The Queue is initialized to be empty. */
{
  front=rear=NULL;
}
```

现在我们开始讨论追加元素的方法，为了在队列的尾部追加一个元素，需要编写 append 函数：

```
Error_code Queue::append(const Queue_entry &item)
/*Post: Add item to the rear of the Queue and return a code of success or return a code of
        overflow if dynamic memory is exhausted. */
{
  Node*new_rear=new Node(item);
  if (new_rear==NULL) return overflow;
  if (rear==NULL) front=rear=new_rear;
  else {
    rear->next=new_rear;
    rear=new_rear;
  }
  return success;
}
```

由于增加一个 Node 到空的 Queue 需要同时设置 front 和 rear 指针以指向新的 Node，而加到非空的 Queue 仅需要改变 rear，所以 Queue 是否为空的情况必须区别对待。

在 Queue 的头部为一个元素提供服务，使用下面的函数：

```
Error_code Queue::serve()
/*Post: The front of the Queue is removed. If the Queue is empty, return an Error_code
        of underflow. */
{
  if (front==NULL) return underflow;
  Node*old_front=front;
  front=old_front->next;
  if (front==NULL) rear=NULL;
  delete old_front;
  return success;
}
```

我们必须再次单独地考虑空 Queue 的可能性。试图从空 Queue 中进行删除操作将产生一个下溢的 Error_code，但 Queue 在进行删除后再变空并不算错误，这时 rear 和 front 应同时变成 NULL 以指示 Queue 已成了空队列。我们将链式队列的其他方法作为练习。

如果将链式队列的这些算法和顺序队列的算法进行比较，你会发现链式方案无论是在概念上还是在程序设计上都要简单一些。Queue 的重载赋值运算符、析构函数和拷贝构造函数的编写均留作练习。

4.4.2　扩展的链式队列

Queue 的链式实现为其他种类的队列类提供了基类，例如第 3 章中定义的扩展队列。下面的C++代码定义了派生类 Extended_queue，它等同于第 3 章中的相应代码。

```
class Extended_queue: public Queue {
public:
  bool full()const;
  int size()const;
  void clear();
  Error_code serve_and_retrieve(Queue_entry &item);
};
```

* * * 默认方法的实现

虽然这个 Extended_queue 类有一种链式实现，但没有必要为拷贝构造函数、重载赋值运算符或析构函数提供显式的方法。编译器为这些方法中的每一个都产生了一个默认的实现，默认的方法将调用基类 Queue 对象的相应方法。例如 Extended_queue 的默认析构函数只是调用链式 Queue 的析构函数，其功能就是删除所有动态分配的 Extended_queue 结点。因为我们的 Extended_queue 存储的链式数据都是类 Queue 已有的部分，因此编译器所产生的默认方法正是我们所需要的。

为了使用基类中的数据成员，需要为链式 Extended_queue 类声明的方法重新设计程序，如新的 size 方法必须使用称作 window 的临时指针去遍历 Queue（换句话说，临时指针沿着队列移动并指向队列中的每个 Node）。

```
int Extended_queue::size()const
/*Post:Return the number of entries in the Extended_queue. */
{
  Node *window=front;
  int count=0;
  while (window!=NULL){
    window=window->next;
    count++;
  }
  return count;
}
```

扩展队列链式实现的其他方法留作练习。

练习 4.4

E1. 编写链式队列的下列方法：

(1) 方法 empty　　(4) 拷贝构造函数

(2) 方法 retrieve　　(5) 重载赋值运算符

(3) 析构函数

E2. 编写 Extended_queue 的 full 方法的一个实现。可以看出这个方法在链式实现里很简单，那么为何还需要将它包含进链式 Extended_queue 类中？

E3. 编写 Extended_queue 链式类的下列方法：

(1)clear　　(2)serve_and_retrieve

E4. 对于链式扩展队列 Extended_queue，由于没有将队列中的元素数作为类的一个独立的成员，因此函数 size 需要用一个循环在整个队列上移动，从而计算出队列中的元素个数。考虑修改链式扩展队列 Extended_queue 的定义，在类中加上一个计数的数据成员。这样需要对类的所有其他方法做出什么修改？与最初的实现相比，讨论这样修改的优缺点。

E5. 图 4.14 所示的循环链表是链表的一种，其表尾结点不是空指针，而是指回到表头结点。因为 tail->next 指回到表头，所以仅需要一个指针 tail 就能够访问表的两端。

(1)如果用循环链表实现队列，那么仅需要一个指针 tail(或 rear)就能定位表头和表尾。编写处理以此方式存储的队列所需要的方法。

(2)与需要两个指针的实现形式相比，实现这种结构有什么缺点？

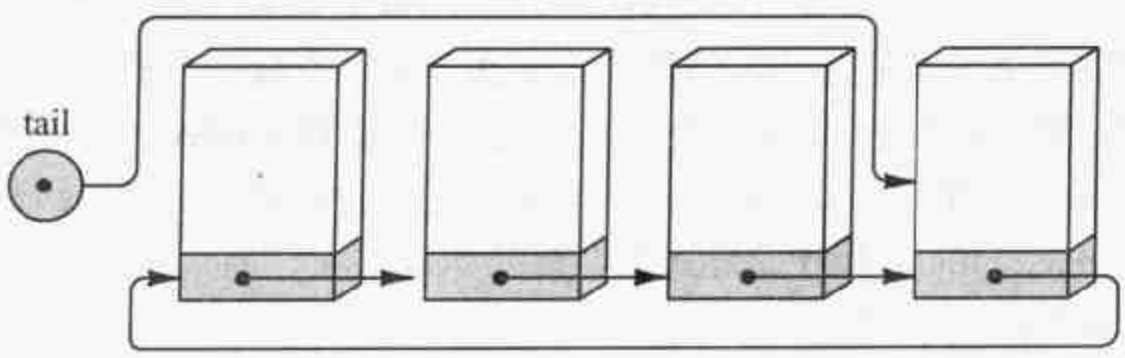


图 4.14　带尾指针的循环链表

程序设计项目 4.4

P1. 汇编链式队列的说明和方法文件，命名为 queue. h 和 queue. c，使得它适合于应用程序使用。

P2. 参考 3.4 节中的字符 Extended_queue 的菜单驱动演示程序，并用链式 Extended_queue 的实现文件替换顺序队列实现文件。如果仔细地设计了其中的程序和类，那么程序不需要进行更多的修改即可正确地运行。

P3. 在 3.5 节开发的飞机场模拟项目中，用链式队列代替其中顺序队列的实现。如果你对类进行过仔细地设计，那么程序不需进行更多的修改即可以同样的方式运行。

4.5　应用：多项式运算

4.5.1　项目的目的

我们曾在 2.3 节设计过一个模拟简单计算器做加、减、乘、除的程序，此程序或许还处理其他的运算。本节的目的是设计一个类似的计算器，但现在的操作对象将是多项式而不是数字。

＊＊＊用于多项式的逆波兰计算器

就像 2.3 节中一样，我们要模拟一个逆波兰计算器，其操作数（此处为多项式）在操作指定前输入，操作数将被压入栈中，操作在执行时，会从栈中弹出操作数，然后将操作结果又压回栈中。我们再次使用 2.3 节的约定（可能你希望复习一下），？表示将操作数压入栈，+，-，＊ 和 /表示算术运算，=表示打印出栈顶元素（不弹出）。例如指令？ a？ b+=意味着读两个操作数 a 和 b，然后计算并打印它们的和。

4.5.2　主程序

显然我们需要实现一个用于计算器的类 Polynomial，这样可以简化设计计算器程序的任务。我们需要定制泛型栈实现以使用多项式元素，然后主程序能够声明一个多项式栈，接受新的命令，并且只要希望就可以执行它们。

```
int main()
/*Post: The program has executed simple polynomial arithmetic commands entered by the
      user.
  Uses: The classes Stack and Polynomial and the functions introduction,
      instructions, do_command, and get_commard . */
{
  Stack stcred_polynomials;
  introduction();
  instructions();
  while (do_command(get_command(), stored_polynomials));
}
```

这个程序几乎与 2.3 节的主程序一样，并且它的辅助函数 get_command 也等同于早期的形式。

1. 多项式方法

*** 用户命令

我们像 2.3 节中一样用字符?，= ，+ ，- ，* ，/表示用户能键入的命令，其中? 表示请求用户输入一个多项式，=打印操作的结果，剩余的符号分别表示加、减、乘、除。

这些命令中的大多数都需要调用 Polynomial 类的方法，因此现在必须确定其中一些方法的形式。

*** 多项式方法

我们需要对两个多项式作加法的方法，实现此方法的一种便利的方式是把 equals_sum 作为 Polynomial 类的一个方法。这样，如果 p，q，r 是 Polynomial 对象，则表达式 p. equals_sum(q, r)用多项式 q 与 r 的和去代替 p。我们将实现类似的方法 equals_difference，equals_product 和 equals_quotient 去执行其他的多项式运算操作。

用户命令 = 和? 将调用 Polynomial 的方法输出和读入多项式，因此我们将假定 Polynomial 对象有不带参数的、称为 print 和 read 的方法去完成这些任务。

2. 执行命令

*** 运行用户命令

按我们先前的决定，可以马上编写函数 do_command。我们提供了这个函数的简写形式，只在它的主 switch 语句中对几种可能性进行编码。

```
bool do_command(char command, Stack &stored_polynomials)
/*Pre: The first parameter specifies a valid calculator command.
  Post: The command specified by the first parameter has been applied to the Stack of
        Polynomial objects given by the second parameter. A result of true is returned
        unless command=='q'.
  Uses: The classes Stack and Polynomial. */
{
  Polynomial p, q, r;
  switch (command) {
  case '?':
    p.read();
    if(stored_polynomials.push(p)==overflow)
      cout<<"Warning: tack full, lost polynomial"<<endl;
    break;
  case '=':
    if(stored_polynomials.empty())
      cout<<"Stack empty"<<endl;
    else {
      stored_polynomials.top(p);
      p.print();
    }
    break;
  case '+':
    if(stored_polynomials.empty())
      cout<<"Stack empty"<<endl;
```

```
    else {
      stored_polynomials.top(p);
      stored_polynomials.pop( );
      if(stored_polynomials.empty()){
        cout<<"Stack has just one polynomial"<<endl;
        stored_polynomials.push(p);
      }
      else {
        stored_polynomials.top(q);
        stored_polynomials.pop();
        r.equals_sum(q, p);
        if(stored_polynomials.push(r)==overflow)
          cout<<"Warning:Stack full, lost polynomial"<<endl;
      }
    }
    break;
  //  Add options for further user commands.
  case 'q':
    cout<<"Calculation finished."<<endl;
    return false;
  }
  return true;
}
```

在这个函数中，我们需要通过引用去传递 Stack 的参数，因为它的值可能需要修改。例如，如果命令参数是＋，那么通常是从栈中弹出两个多项式并且将它们的和入栈。函数 do_command 也允许附加的用户命令 q 用来退出程序。

3. 占位程序和测试

现在已设计了足够的程序，可以暂停一下，先去编译、调试和测试这些程序，确保到目前为止所做的是正确的。

为了编译程序，当然需要为所有缺少的成分补充占位程序。由于在编译时可以使用早期的栈实现，所以目前惟一缺少的部分是类 Polynomial，但我们目前还没有决定如何存储多项式对象。

＊＊＊临时类型声明

作为测试，我们将程序作为普通逆波兰计算器对实数进行运算，因此需要一个使用实数的占位类声明去代替多项式。

```
class Polynomial {
public:
  void read();
  void print();
  void equals_sum(Polynomial p, Polynomial q);
  void equals_difference(Polynomial p, Polynomial q);
  void equals_product(Polynomial p, Polynomial q);
  Error_code equals_quotient(Polynomial p, Polynomial q);
private:
  double value;
};
```

由于 equals_quotient 方法必须检测被 0 除的错误，因此它有一个 Error_code 的返回类型，其他的方法不去检测错误，因此有 void 的返回类型。下面的函数是所需要的典型的占位方法。

```
void Polynomial::equals_sum(Polynomial p, Polynomial q)
{
  value=p.value+q.value;
}
```

在此时生成程序的框架保证了栈和实用程序包也能恰当地并入到程序中。无论我们使用顺序栈还是链栈，此程序及其占位程序都应该正确地运行。

4.5.3 多项式数据结构

*** 多项式的实质

现在开始完成我们的基本任务，即确定如何表达多项式并编写方法去操作它们。如果我们仔细考虑如下的多项式：$3x^5-2x^3+x^2+4$。我们可以看到多项式的重要信息都保存在 x 的系数和指数中，变量 x 本身实际上仅是一个占位符(哑变量)。因此，为计算目的，我们认为多项式是由项组成的，每一项又由一个系数和一个指数组成。在计算机里，我们能类似地将多项式表示为系数和指数对的表，其中的每一对构成一个结构，我们称它为 Term，下面以带构造函数的 struct 去实现 Term：

```
struct Term {
  int degree;
  double coefficient;
  Term(int exponent=0, double scalar=0);
};
Term::Term(int exponent, double scalar)
/* Post: The Term is initialized with the given coefficient and exponent, or with
         default parameter values of 0. */
{
  degree=exponent;
  coefficient=scalar;
}
```

在将多项式表示为由项构成的表之后，我们必须加入自己的方法规则，用来在这样的两个表上执行运算。但是当我们做这个工作时，就会发现需要不断地从表中删除第一个元素，并且仅仅需要在表尾插入新的元素。换句话说，我们发现运算操作把表当作队列，或更精确地说，是当作扩展队列，因为我们经常需要 clear 和 serve_and_retrieve 之类的方法，也需要从队头删除和在队尾插入。

*** 多项式的实现

我们该使用顺序队列还是链式队列？如果我们事先知道多项式的次数并且如果多项式几乎所有的项的系数都不为零，则使用顺序队列可能更好些。但是如果我们不知道次数的边界或者如果多项式仅可能出现少数非零项，那么链式存储将更可取。实际上我们决定将多项式表示为项的扩展链式队列，这种表示在图 4.15 中说明。

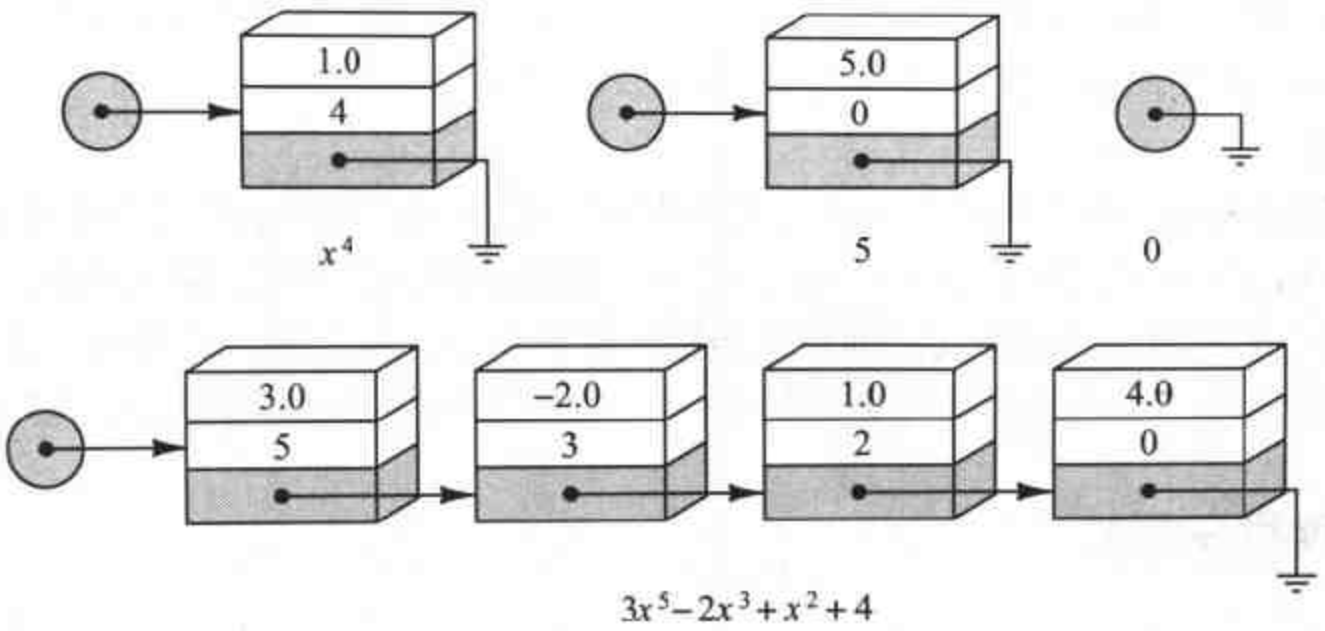


图 4.15　多项式表示为项的链式队列

＊＊＊假设

每个结点包含多项式的一项，并且仅仅在队列中保存非零的项。总为零的多项式(即仅有一个 0 项)用空队列表示，我们称它为零多项式或说它等于 0。

我们关于 Polynomial 数据结构的决定暗示了我们要将它当作扩展队列的派生类来实现，这样就可以重用 Extended_queue 的操作，而将注意力集中在多项式特有的附加方法的编码实现上。

在实现这样的派生类之前，我们将进行最终的检查，需要提出的问题是：Polynomial 是一个 Extended_queue 吗？

Extended_queue 允许像 serve 这样不直接应用于多项式的方法，因此我们必须承认 Polynomial 不是真正的 Extended_queue。(在编码实现时，如果我们要努力防止客户程序从 Polynomial 对象中删除元素，那么这个缺陷将变得更明显。)因此，虽然在实现 Polynomial 类时，重用 Extended_queue 类的数据成员和函数代码将很有用，但由于这两个类之间并没有表现出 is—a 关系，所以我们不采用这种简单的继承实现。

＊＊＊私有继承

C++语言提供了另一种形式的继承，称为私有继承，这正是我们所需要的。私有继承模拟了类之间“用……来实现”的关系，因此我们定义 Polynomial 类是从 Extended_queue 私有继承的，这意味着 Extended_queue 的成员和方法在 Polynomial 类的实现里是可用的，但它们对于使用 Polynomial 的客户程序是不可用的。

```
class Polynomial: private Extended_queue {   // Use private inheritance.
public:
  void read();
  void print() const;
  void equals_sum(polynomial p, Polynomial q);
  void equals_difference(Polynomial p, Polynomial q);
  void equals_product(Polynomial p, Polynomial q);
  Error_code equals_quotient(Polynomial p, Polynomial q);
  int degree() const;
private:
  void mult_term(Polynomial p, Term t);
};
```

我们包含了一个有用的方法 Polynomial::degree(),它返回 Polynomial 里主项的次数。同时还有一个辅助函数,它用单个 Term 去乘多项式 Polynomial。

* * * 限制

我们目前为止还没有考虑多项式项的存储顺序,如果允许它们以任何顺序存储,那么会很难识别,如 x^5+x^2-3 和 $-3+x^5-x^2$ 和 x^2-3+x^5 都表示同一多项式。因此我们采用通常的约定,即每个多项式的项在链式队列里以指数的降序存储。我们进一步假设不存在具有同样指数的两个项,并且也不存在零系数的项。(记住等于 0 的多项式表示为空队列。)

4.5.4 读写多项式

* * * 标准的约定

以链式队列去实现多项式时,写出多项式是一件简单的事,只要在队列的结点上循环并输出每个结点的数据。下面的 print 方法的复杂特性只是习惯的反映,是用于书写多项式的特殊约定,而不是对我们的数据结构进行处理所产生的任何概念上的困难。特别地,我们的方法取消了任何初始的"+"号、值为 1 的系数和指数以及任何涉及 x^0 的引用,因此我们会仔细地输出 $3x^2+x+5$ 和 $-3x^2+1$ 面不是 $+3x^2+1x^1+5x^0$ 和 $-3x^2+1x^0$。

```
void Polynomial::print()const
/*Post: The Polynomial is printed to cout. */
{
  Node*print_node=front;
  bool first_term=true;
  while (print_node!=NULL){
    Term &print_term=print_node->entry;
    if(first_term){          //  In this case, suppress printing an initial '+ '.
      first_term=false;
      if (print_term.coefficient<0)cout<<"-";
    }
    else if (print_term.coefficient<0)cout<<"-";
    else cout<<"+";
    double r=(print_term.coefficient>=0)
             ? print_term.coefficient:-(print_term.coefficient);
    if (r!=1)cout<<r;
    if (print_term.degree>1)cout<<"X^"<<print_term.degree;
    if (print_term.degree==1)cout<<"X";
    if (r==1 && print_term.degree==0)cout<<"1";
    print_node=print_node->next;
  }
  if(first_term)
    cout<<"0";           //  Print 0 for an empty Polynomial.
  cout<<endl;
}
```

当读入新的多项式时,我们将构造一个新的 Polynomial 对象,然后为输入中读取的每个项(系数和指数对)追加一个元素到对象中。

就像从用户那里直接接受输入的所有函数一样,读取新多项式的函数必须仔细检查它的输入,确保它满足问题的需求。确保多项式里的指数以降序出现是函数的重要任务之一,

为此，需要不断地将当前项的指数同前一个项的指数进行比较。

我们将使用系数为 0.0 或者指数为 0 的特殊值去终止读入的过程：以 0.0 为系数的项永远不会存储在多项式中，而由于指数以降序排列，那么任何具有 0 指数的项总会排在最后。生成的函数如下：

```
void Polynomial::read()
/* Post: The Polynomial is read from cin. */
{
  clear();
  double coefficient;
  int last_exponent, exponent;
  bool first_term=true;
  cout<<"Enter the coefficients and exponents for the polynomial,"
      <<"one pair per line.Exponents must be in descending order."<<endl
      <<"Enter a coefficient of 0 or an exponent of 0 to terminate."<<endl;
  do {
    cout<<"coefficient?"<<flush;
    cin >>coefficient;
    if(coefficient!=0.0){
      cout<<"exponent?"<<flush;
      cin >>exponent;
      if((!first_term && exponent>=last_exponent)||exponent<0){
        exponent=0;
        cout<<"Bad exponent:Polynomial terminates without its last term."
            <<endl;
      }
      else {
        Term new_term(exponent, coefficient);
        append(new_term);
        first_term=false;
      }
      last_exponent=exponent;
    }
  } while (coefficient!=0.0 && exponent!=0);
}
```

4.5.5　多项式加法

现在来研究多项式的一种基本操作，即将两个多项式相加。

要求 Polynomial 的项在相应 Extended_queue 中以指数的降序出现大大简化了其加法操作。此时为了将两个多项式相加，只需要每次一个地扫描它们，如果发现两个多项式有同样指数的项，则将它们的系数相加；否则，只需将指数较大的项复制到和中，并移至多项式中的下一项。在此过程中，注意不要将 0 系数的项放入和里。

```
void Polynomial::equals_sum(Polynomial p, Polynomial q)
/*Post: The Polynomial object is reset as the sum of the two parameters. */
{
  clear();
  while (! p.empty()||!q.empty()){
    Term p_term, q_term;
```

```
    if(p.degree()>q.degree()){
      p.serve_and_retrieve(p_term);
      append(p_term);
    }

    else if (q.degree()>p.degree()){
      q.serve_and_retrieve(q_term);
      append(q_term);
    }

    else {
      p.serve_and_retrieve(p_term);
      q.serve_and_retrieve(q_term);
      if(p_term.coefficient+q_term.coefficient!=0){
        Term answer_term(p_term.degree,
                         p_term.coefficient+q_term.coefficient);
        append(answer_term);
      }
    }
  }
}
```

这个方法首先清除掉用于记录答案的Polynomial对象中当前存储的所有项。可以用一个循环完成加法任务，每次从一个或两个多项式参数中摘取第一项并将这些项加入答案，需要根据参数的次数决定由哪个或哪些参数提供下一项。

Polynomial的次数由方法degree()计算，它用retrieve()方法取得多项式的主项并返回其次数。我们遵循标准的数学约定，将零多项式的次数赋值为-1。

```
int Polynomial::degree()const
/*Post: If the Polynomial is identically 0, a result of -1 is returned. Otherwise the
        degree of the Polynomial is returned. */
{
  if (empty())return -1;
  Term lead;
  retrieve(lead);
  return lead.degree;
}
```

4.5.6 完成项目

1. 缺少的过程

此时，Polynomial类的剩余方法非常类似于那些已经编写的方法，因此将它们都留作项目。剩余的运算操作方法在总的形式上和equals_sum相同，其中的一些方法比较简单：减法几乎等同于加法。对于乘法，我们可以先编写一个单项乘以Polynomial的函数，然后组合使用这个函数和加法函数去实现乘法。除法则更复杂。

2. 栈实现的选择

Polynomial类的实现利用了多项式项的链式Extended_queue，因此，我们必须声明

Node 中包含有 Term 作为元素，这样就妨碍我们使用链式 Stack 类去包含 Polynomial 元素（因为那样会需要包含 Polynomial 元素的结点），为此必须用顺序 Stack 的实现去编译我们的计算器程序。

* * * 模板

这是由于我们对泛型的简单处理而造成的第一个不利的情形。但是就像我们以前所看到的，C++确实为使用模板的泛型提供了更先进的方法。如果我们在整章都系统地使用了模板，那么我们的计算器程序就可以用链式或顺序 Stack 实现编译。在下一章，我们将开始使用模板去实现真正的泛型数据结构。

3. 团体项目

用于操作多项式的相干函数包的成果构成一个有趣的团体项目，团体中的不同成员能为不同的操作编写辅助函数或方法，其中的一些操作被留作本节末尾的项目，但是你可能希望同时包含附加的功能，但对任何附加的特征都要经过仔细规划，确保它们能在合理的时间内完成，不会破坏程序的其他部分。

* * * 说明

在确定了成员之间的工作划分后，与团体有关的最重要的决定是函数和方法彼此之间通信的正确方式，特别是它们与调用程序之间的通信。如果希望对程序的组织做出任何修改，一定要确信清楚完整地说明团体中所有成员的精确的细节。

* * * 合作

其次，如果期望团体成员在同一时间完成他们的工作，或者期望项目的所有部分能够组合且一起调试，这样的要求未免太高。在此过程中，我们需要使用程序占位和驱动器（见 1.4 节）去调试和测试项目的各个部分。无论如何，你会发现让不同的成员互相读取、帮助调试和测试彼此的功能是非常有效的手段。

* * * 协调

最后，我们要负责确保团体的所有成员及时完成各自的工作，跟踪项目在各个方面的进展，确保函数在完全编译和测试通过之前不会被合并到项目中，并将所有的工作组合成最终的成果。

练习 4.5

E1. 讨论如何扩展多项式计算器，使它能处理多变量多项式。

程序设计项目 4.5

P1. 汇编在本节中开发的函数，对其中的代码做出必要的修改，从而生成计算器程序的运行框架，它可以读、写和相加多项式。需要补充函数 get_command()，introduction()和 instructions()。

P2. 编写 Polynomial 的方法 equals_difference 并将它集成到计算器中。

P3. 编写辅助函数 void Polynomial :: mult_term(Polynomial p, Term t)，它通过用单个的

Term t 乘以 p 来计算 Polynomial 对象。

P4. 用前一问题中开发的函数和 Polynomial 的方法 equals_sum，编写 Polynomial 的方法 equals_product，并将最后得到的函数集成到计算器中。

P5. 编写 Polynomial 的方法 equals_quotient 并将它集成到计算器中。

P6. 许多逆波兰计算器不仅使用栈，而且提供了操作数能存储的内存位置。对项目进行扩展，提供数组用来存储多项式。提供另外的命令以将栈顶存储到一个数组元素中并将数组元素里的多项式压入栈中。此数组应有 100 个元素，并且 100 个位置都应当在程序开始时初始化成 0 多项式，访问数组的函数应询问用户使用哪一个元素。

P7. 编写函数丢弃栈顶的多项式，并将此能力包含为一个新的命令。

P8. 编写函数交换栈的最顶部的两个多项式，并将此能力包含为一个新的命令。

P9. 编写函数将栈中的所有多项式相加，并将此能力包含为一个新的命令。

P10. 编写函数计算多项式的导数，并将此能力包含为一个新的命令。

P11. 编写函数，给定一个多项式和一个实数，求多项式在此数上的值，并将此能力包含为一个新的命令。

P12. 编写一个新的方法 equals_remainder，它得到第一个 Polynomial 参数被第二个 Polynomial 参数除的余数，在计算器程序中增加一个新的用户命令 % 去调用该方法。

4.6 抽象数据类型及其实现

当我们首次介绍栈和队列时，我们仅仅认为它们是在顺序存储中实现的，而在介绍链栈和链式队列时，去识别同样的基础抽象数据类型时我们没有遇到困难。为了阐明从抽象数据类型定义到C++实现的一般过程，我们仔细考虑这些数据类型和已经看到的实现。

我们从回忆 2.5 节中栈 ADT 的定义开始。

定义 元素类型为 T 的栈是 T 的元素的一个有限序列，同时带有如下操作：

1. 创建一个栈，使它为空(create)。
2. 测试栈是否为空(empty)。
3. 倘若栈不满，则将一个新元素压入栈顶(push)。
4. 倘若栈不空，则从栈顶弹出一个元素(pop)。
5. 倘若栈不空，则得到栈顶元素(top)。

为了得到队列 ADT 的定义，我们用如下的队列方法替换栈的方法。

定义 元素类型为 T 的队列是 T 的元素的一个有限序列，同时带有如下的操作：

1. 创建队列，使它为空(create)。
2. 测试队列是否为空(empty)。
3. 倘若队列不满，则将一个新元素追加到队尾(append)。
4. 倘若队列不空，则从队头删除元素(serve 和 remove)。
5. 倘若队列不空，则得到队头元素(retrieve)。

我们也能得到如下扩展队列的精确定义。

定义　元素类型为 T 的扩展队列是 T 的元素的一个队列，同时带有如下的附加操作：

4. 判断队列是否已满(full)。
5. 得到队列的长度(size)。
6. 若队列不空，删除和得到队头元素(serve 和 retrieve)。
7. 清除队列使它为空。

注意这些定义并没有提到抽象数据类型(栈、队列或扩展队列)的实现方式。在前面几章里，我们已经学习了每种类型的不同实现，而这些新的定义同样符合其中任何一种实现。

回忆一下 2.5 节，在实现抽象数据类型的过程中，必须从类型定义的抽象层开始，通过数据结构层，再到实现层。其中数据抽象层模拟数据类型的结构，实现层决定如何在计算机内存中存储数据结构的细节。图 4.16 以队列的情况举例说明了这些阶段的细化。我们从序列的数学概念开始，将队列看成一种抽象数据类型。在接下来的一层，我们从图中显示的各种数据结构中选择，从物理模型(在这个模型中当每一项离开队头时，所有的项都向前移动)到线性模型(在这个模型中队列一起被置空)，再到循环数组，最后到链表。就像在下一层中显示的，其中的一些数据结构允许在实现上有更多的变化。在最后阶段，为一个特定的应用对队列进行编码。

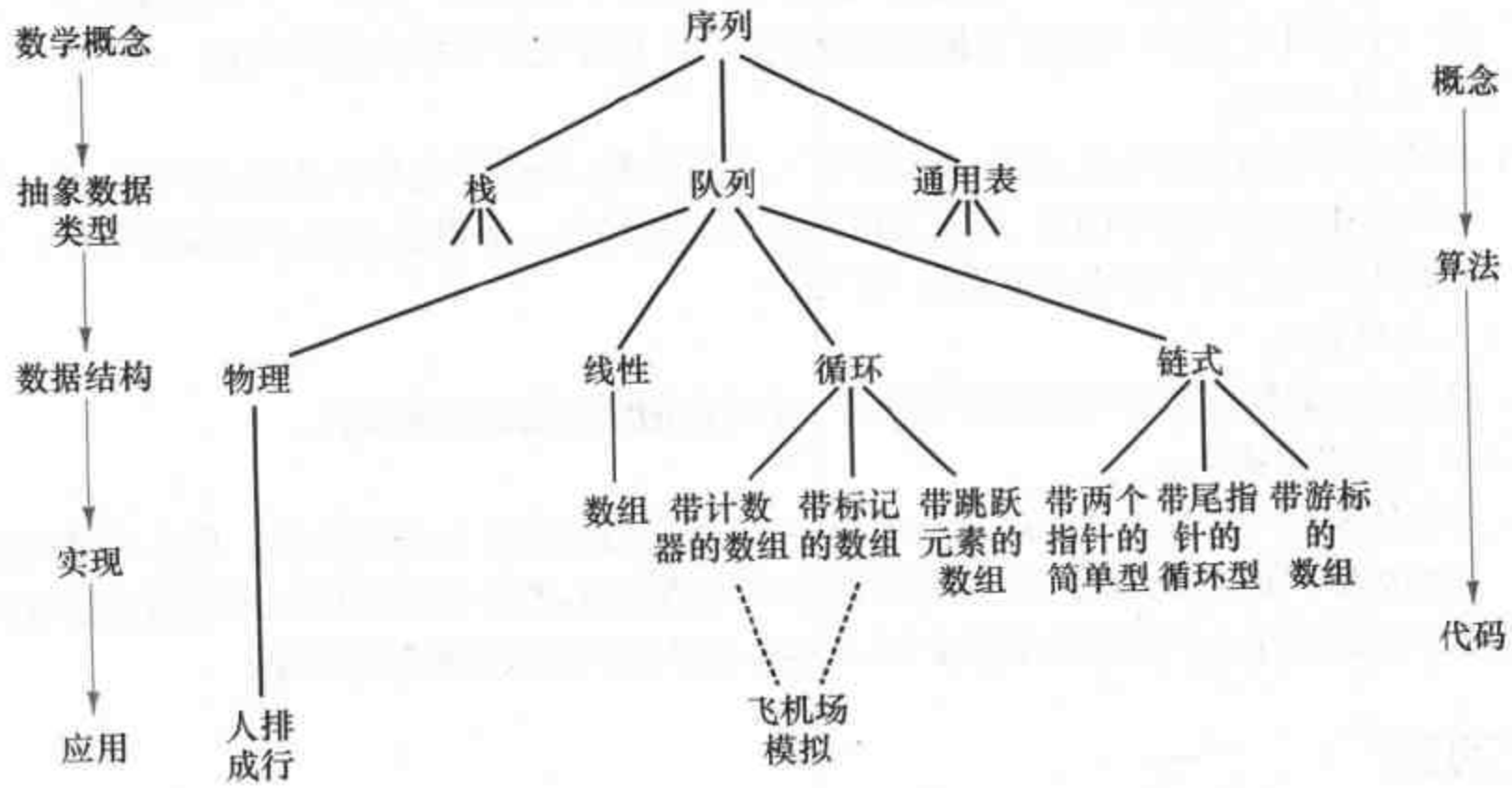


图 4.16　队列的细化

练习 4.6

E1. 类似于图 4.16 所示的细化层次，为栈画一个图。

E2. 以给出的栈和队列的定义为模型，给出术语 deque 的形式化定义。回忆元素可以从双端队列的任一端加入或删除，但只限于在端部进行。

启示和易犯的错误

1. 在选择实现前，确信已在抽象层充分说明所有的数据结构及其相关操作。
2. 在选择链式和顺序实现时，考虑数据结构上的必要操作。链式结构在插入、删除和重排操作中更灵活，而顺序结构有时候更快。
3. 当结构中的项较小且算法简单时，顺序结构一般需要较少的内存、计算时间和程序设计工作。当结构存放较大的记录时，链式结构通常更节省空间、时间和程序设计工作。
4. 动态内存和指针允许程序自动地适应大范围的应用规模，而且在不同数据结构的空间分配上提供了灵活性。自动内存对完全可以预先描述规模的应用有时更有效。
5. 在对指针重新赋值前，确信它所引用的对象不会变成垃圾。
6. 将未初始化的指针设置为 NULL。
7. 链式数据结构应当实现析构函数、拷贝构造函数和重载赋值运算符。
8. 使用私有继承去模拟类间的“用……实现”的关系。
9. 画出链式结构适当部分的“之前”和“之后”图，显示相关指针及其应被修改的方式。如果有帮助的话，也画图显示此过程的中间阶段。
10. 确定值应以何种顺序放入指针域以实现各种修改，通常最好先将值赋给先前未定义的指针，然后赋给值为 NULL 的指针，最后再赋给剩下的指针。在将一个指针变量复制到另一个指针变量后，可以自由地重新分配第一个指针变量以指向新的位置。

*** 未定义的链

11. 在链式结构的方法结束之际，确信没有未定义的链，如新结点中从未赋值的链，或者是旧结点中已变为悬浮的链，即它指向不再使用的结点。应该将这样的链重新分配给仍旧在用的结点或者将它的值设为 NULL。

*** 极端情况

12. 总是验证你的算法对空的结构和仅有一个结点的结构能正确地工作。

*** 多重引用解除

13. 避免使用(p->next)->next 这样的构造，即使它们在语法上是正确的。单个对象应仅仅包括单个的指针引用解除，对重复解除引用的结构，通常预示着可以通过重新思考算法中应声明什么样的指针变量来加以改进，如果有必要就引进新的变量。

复习题

4.1

1. 为什么动态内存分配是重要的？给出两个原因。
2. 什么是垃圾？
3. 为什么未初始化的指针应设置为 NULL？
4. 什么是别名，为什么它是危险的？

4.2

5. 为什么链式 Stack 的 push 方法要返回一个 Error_code？

4.3

6. 为什么我们总要为链式数据结构加上析构函数?

7. 如何使用构造函数? 为什么链式数据结构要包含拷贝构造函数?

8. 为什么链式数据结构应采用重载赋值运算符实现?

4.5

9. 讨论在团体程序设计项目中可能发生而在个人程序设计项目中不会发生的一些问题,团体项目较个人项目有什么有利之处?

4.6

10. 抽象数据类型在多大程度上说明了实现?

11. 列举(按抽象到具体的次序)数据说明的 4 个细化层次。

第5章 递 归

本章介绍递归，这种解决问题的方法是将问题简化为同样问题的更小的子问题。为了说明递归方法，我们将研究一些应用和样例问题，从而证明递归方法可以富有成效地解决某些类型的问题。其中的一些例子比较简单，另一些例子则相当复杂。本章同时也分析如何在计算机上实现递归。在此过程中，将得出关于递归的正确使用和不正确使用、何时适合使用递归以及何时最好避免使用递归等方面的指导原则。

5.1 递归导言

5.1.1 子程序的栈帧

＊＊＊调用记录

作为栈的一种重要应用，考虑函数被调用时计算机系统内会发生什么。系统（或程序）必须记住发出调用的位置，以便函数完成后能够返回。它也必须记住所有的局部变量、处理器的寄存器等，以便函数运行时信息不会丢失。我们可以将所有这些信息想像成一种大型的数据结构，每个函数有一个临时存储区，这种结构有时被称为函数调用的调用记录或激活记录。

＊＊＊嵌套的函数调用

假设现在有三个函数A，B和C，并假设A调用B且B调用C。那么在C结束并返回前，B不会结束运行。类似地，A最先开始运行，但它直到B结束并返回后的某个时间才会结束，在三个函数中它最后结束。因此函数活动发生的序列可以概括成“后进先出”性质。如果考虑机器安排临时存储区域供函数使用的任务，则这些区域将会在具有同样性质的一个序列中进行分配；也就是说，是在一个栈中分配（见图5.1，其中M代表主程序的调用记录，A，B和C分别代表相应函数的调用记录）。因此，栈在计算机系统的函数调用中起着关键作用。

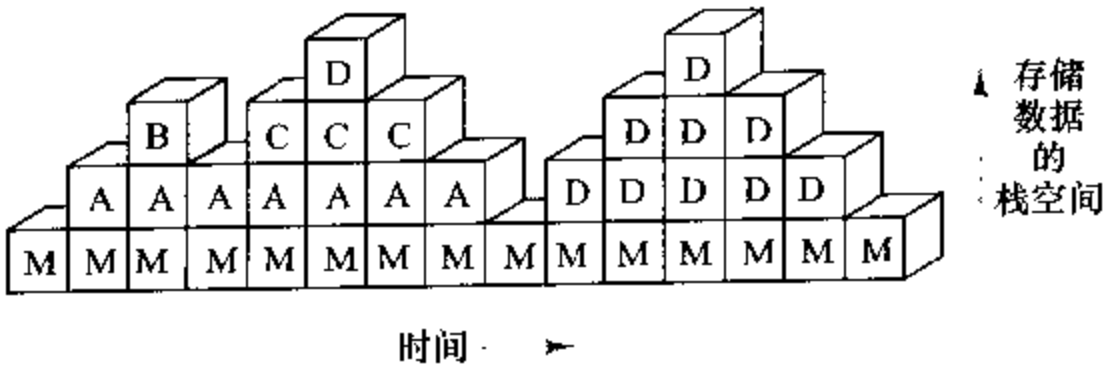


图5.1 函数调用的栈帧

* * * 栈帧

* * * 定义：递归

图 5.1 展示了一个栈帧序列，其中每个垂直的列是某给定时刻栈的内容，通过自左向右通读可以描绘栈的变化。从图 5.1 我们注意到，压入栈中的临时存储区是来自不同函数还是来自同一函数的重复事件是没有区别的。递归这个名称用于函数调用了自己，或者函数调用了一系列其他函数，而其中的一个函数最终又再次调用了第一个函数的情况。关于函数调用的栈帧，递归与任何其他的函数调用并无区别。

5.1.2 子程序调用树

另一个图说明了栈和函数调用之间的关系。这种“树”图展示了函数被调用的顺序。图 5.2 中给出的树图与图 5.1 中所示的栈帧对应。

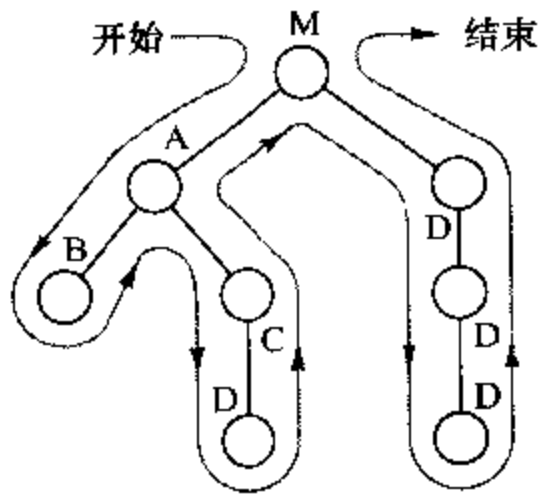


图 5.2 函数调用树

* * * 根、顶点和结点

我们从树顶开始。树的顶部称为根，对应于主程序。每个圆（称为顶点或结点）对应于函数的一次调用。主程序直接做出的所有调用直接由根下的顶点显示。当然每个函数也能调用其他的函数，这些被调用的函数由低层更深的顶点显示。这样，这棵树就长成如图 5.2 中所示的树那样，我们称这样的树为函数调用树。

* * * 孩子和双亲

* * * 分支、兄弟和树叶

* * * 高度、深度和层次

我们经常会用到其他几个关于树的术语，冒昧地借鉴了生物学上的树和族树。直接位于给定顶点下面的所有顶点称为给定顶点的孩子，其上面的直接顶点（惟一的）称为其双亲。将顶点同直接在它上面或下面的顶点连接起来的线段称为分支。兄弟是指那些拥有相同双亲的顶点。树根（根结点）是树中惟一没有双亲的顶点。没有孩子的顶点称为树叶，或者有时也称为外部顶点。例如在图 5.2 中，M 是根，A 和 D 是其孩子，B 和 C 是 A 的孩子，B 和最底部出现的两个 D 是树叶（D 的另外两个出现不是树叶）。如果一棵分支的下部的顶点是另一棵分支的上部的顶点，则称这两棵分支邻接。在一个分支序列中，如果每个顶点都与其后继顶点邻接，则它为一条路径。树的高度是从根到树叶的具有最长可能的路径上的顶点数目。因此图 5.2 中的树的高度为 4，而仅包含一个顶点的树的高度为 1。有时（但并非对

所有函数调用)我们允许空树(无顶点),空树的高度为0。顶点的深度或层是从树根到这个顶点的路径上的分支的数目,因此树根的深度为0。在图5.2中,A的深度为1,B和C的深度为2。

＊＊＊遍历

为跟踪程序中所做的函数调用,我们可以从树根开始并围绕树移动,就像图5.2中的有色路径。这种浅灰色路径称为树的遍历。在向下移动的过程中当我们到达一个顶点时,即调用了这个函数。在遍历了此顶点下面的树以后,我们会在向上的路上再次到达此顶点,这代表终止这个函数并从此函数返回。树叶代表不调用任何其他函数的函数。

＊＊＊递归树

＊＊＊执行跟踪

我们对递归特别感兴趣,因此通常仅提取出树中展示递归调用的部分,并称它为递归树。从图中来看,首先应注意的是递归调用出现的方式与其他函数调用发生的方式没有区别。不同的递归调用仅以不同顶点出现,但却有同样的函数名称。其次,仔细注意,树展示了对函数的调用。因此在多次执行的循环中,虽然函数只出现一次,但它会在树中出现多次,其中每一次出现对应循环中的一次函数执行。类似地,如果在条件语句中的调用函数未被执行,则此调用亦不会出现在树中。

＊＊＊栈帧

＊＊＊空间需求

图5.1中的栈帧展示了递归调用的嵌套,并说明了递归的存储要求。如果某函数递归调用自己数次,则每次递归调用都要为函数中所定义的变量创建单独的副本。在通常的递归实现中,这些数据保存在栈中。注意这个栈所需要的空间数量与递归树的高度成比例,而不是与此树中的全部结点数成比例。即,实现一个递归函数所需要的空间数量依赖于递归的深度,而不是函数被调用的次数。

实际上,后面两个图可以在比调用函数过程更广的上下文中加以解释,因此它们说明了虽然简单但却重要的观察结果,给出了任意的树和栈之间的一种密切关系。

定理5.1 在遍历任何树的过程中,以栈的方式将顶点加入,并在返回根的路径上删除。反之,给定任意的栈,当元素被压入栈或从栈中弹出时,可以画出一棵树来描绘栈的生存史。

现在开始研究递归的几个简单例子,下一步分析一般如何在计算机上实现递归。在此过程中,我们将得出关于递归的正确使用和不正确使用、何时适合使用递归以及何时最好避免使用递归等方面的指导原则。本章其余部分包含递归的几个更为复杂的应用。

5.1.3 阶乘:一种递归定义

＊＊＊非形式化定义

在数学中,一个正整数的阶乘函数一般由下式定义:

$$n! = n \times (n-1) \times \cdots \times 1$$

＊＊＊形式化定义

公式中的省略号(三个点)的意义是“以同样方式继续”。此符号并不精确,因为可能有多种敏感的方式满足此省略号的需要。为计算阶乘,我们需要另一种更加精确的定义,如下:

$$n! = \begin{cases} 1 & \text{若 } n = 0 \\ n \times (n-1)! & \text{若 } n > 0 \end{cases}$$

倘若我们仔细遵循此规则并用一张纸帮助记住进行到何处,那么这种定义能准确地告诉我们如何计算一个数的阶乘。

＊＊＊示例

假设希望计算 4!。由于 4>0,则由定义得到 $4! = 4 \times 3!$。但这还不够充分,再次利用定义,我们得到 $3! = 3 \times 2!$。现在还是不知道 2! 的值,但由定义得到 $2! = 2 \times 1!$。我们仍然不知道 1!,但由于 1>0,因此 $1! = 1 \times 0!$。最后,定义单独处理 $n=0$ 的情况,给出 $0! = 1$。现在可以将这个答案替换进计算 1! 的表达式,并得到 $1! = 1 \times 0! = 1 \times 1 = 1$。现在知道为什么要用一张纸来跟踪部分结果了。除非我们及时将此计算结果写下来,否则在我们到达最底层并用这些结果来完成更早的计算之前就会将此过程早先的步骤给忘掉了。对阶乘计算,有组织地写出所有步骤当然很简单:

$$\begin{aligned} 4! &= 4 \times 3! \\ &= 4 \times (3 \times 2!) \\ &= 4 \times (3 \times (2 \times 1!)) \\ &= 4 \times (3 \times (2 \times (1 \times 0!))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) \\ &= 4 \times 6 \\ &= 24 \end{aligned}$$

＊＊＊问题简化

这个计算阐明了递归工作方式的本质。为了获得一个大型问题的答案,用一种通用方法将大型问题简化成一个或多个具有类似性质但规模更小的问题。然后对这些子问题使用同样的通用方法,因此使递归持续,直到子问题的规模简化到一种最小的基本情况,可以直接给出解答而不必再使用进一步的递归。换言之:

＊＊＊递归的特征

每个递归过程包括两个部分:

1. 不需用递归处理的最小的、基本的问题;
2. 将特定问题简化成一个或多个更小的问题的通用方法,由此前进,最终将问题一直简化成基本问题。

C++(与其他多数现代计算机语言一样)提供了使用递归的简单方式,在 C++ 中阶乘函数变成如下的函数:

* * *递归程序

```
int factorial(int n)
/* Pre:  n is a nonnegative integer.
   Post: Return the value of the factorial of n.*/
{
   if (n==0)
      return 1;
   else
      return n *factorial(n-1);
}
```

从这个计算阶乘的例子中可以看出，问题的递归定义和递归解答可能都显得简洁而优雅，但计算细节却要求在此过程完成前跟踪许多局部计算。

* * *记住局部计算

计算机可以用一个栈轻易地跟踪这种局部计算，但人脑却根本不适合这种任务。让人来记住一长串局部结果并回过头来完成此工作是极其困难的。例如，考虑下面的童谣：

在我去圣伊斯的路上，
我遇到一个带着七个妻子的人。
每个妻子带着七个布袋，
每个布袋里有七只猫，
每只猫带着七套用具，
用具，猫，布袋和妻子，
有多少去圣伊斯？

人很难同时跟踪许多局部计算，因此当我们使用递归时，有必要用略微不同的术语而不是用程序设计的方法来进行思考。程序设计员必须着眼于大的方面，而将细节计算留给计算机去完成。

我们必须在算法中详细说明将大型问题简化成更小的问题的一般步骤的精确形式，必须确定停止规则（最小问题）及其处理方式。另一方面，除了一些较小的简单例子外，我们不打算为了理解递归算法而将一般问题一直进行到停止规则，或者去跟踪计算机对大型问题的操作，否则我们很快就会被所有积压的任务弄糊涂，从而失去有关完整问题和总体解答方法的线索。

现在已有一些很好的通用方法和工具，使我们能够将注意力集中在通用方法和关键步骤上，并同时分析计算机执行所有细节时所做的工作量。我们现在介绍一个例子，它阐明了其中的一些方法和工具。

5.1.4 分而治之：汉诺塔

1. 问题

* * *故事

19世纪，在欧洲出现了一个名为汉诺塔的故事，附带的材料（毫无疑问是杜撰的）解释说这个游戏代表了梵天寺正在进行的一项任务。在创世时，神父得到一个带有3根金刚针的铜台，在第1根针上堆叠了64个金盘，每一个都比自己下面的那个略小。（在欧洲流传的

版本没有那么奇异，只有 3 根木杆和 8 片纸板。）分配给神父的任务是将所有金盘从第 1 根针移到第 3 根针上，需要服从的规则是每次只能移动一个盘并且不允许将盘子放在比它更小的盘子上面。这位神父被告知当他们完成 64 个盘的移动的时候，就意味着世界末日。见图 5.3。

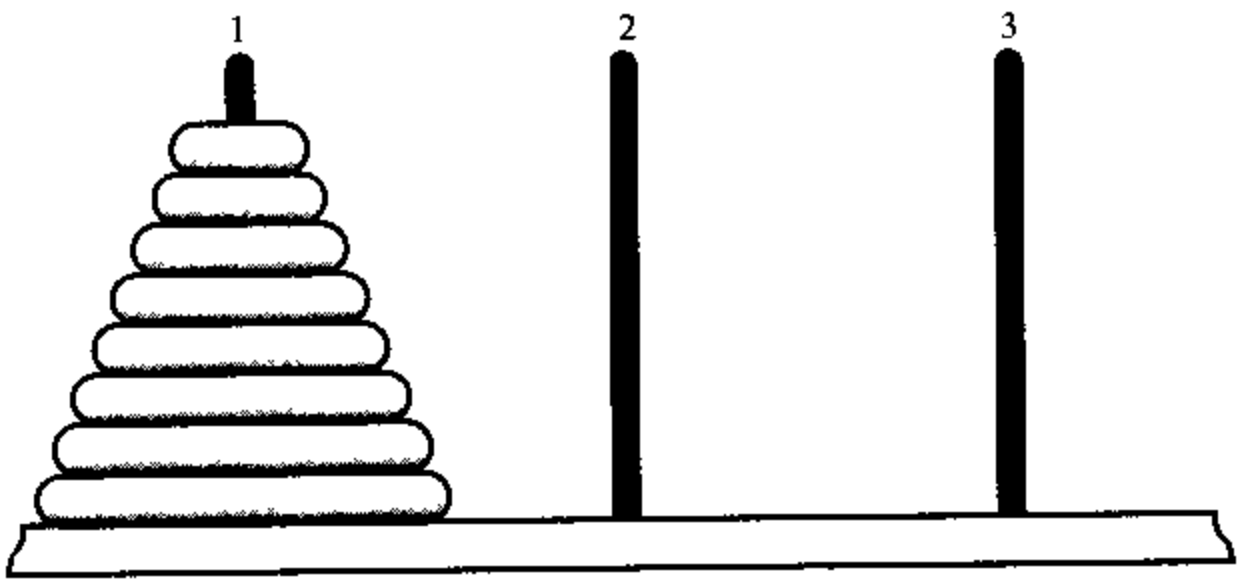


图 5.3　汉诺塔

我们的任务当然是编写一个计算机程序为神父打印出指令列表。可以将这个任务概括为指令 move(64，1，3，2)。它的意义是：利用塔 2 作为临时存储，将 64 个盘从塔 1 移到塔 3。

2. 解答

解答的思想是不要将注意力集中在第 1 步（必然是将最顶层的盘子移到某处），而是要集中在最困难的一步：移动底部的盘子。而在上面的 63 个盘子被移走之前，没有办法到达底部的盘子。并且这些盘子必须全部放在塔 2 上，这样我们才能将底部的盘子从塔 1 移到塔 3。这是因为每个时刻只能移动一个盘子，并且底部的（最大的）盘子决不能放在任何其他盘子的上面。因此当我们动手移动底部的盘子时，在塔 1 和塔 3 上不会再有其他的盘子。因此可以将汉诺塔问题的算法步骤概括成：

```
move(63, 1, 2, 3);      // Move 63 disks from tower 1 to 2 (tower 3 temporary).
cout<< "Move disk 64 from tower 1 to tower 3." <<endl;
move(63, 2, 3, 1);      // Move 63 disks from tower 2 to 3 (tower 1 temporary).
```

＊＊＊通用简化

我们现在向解答迈出了一小步，这只是非常小的一步，因为我们必须再次描述如何移动这 63 个盘子。尽管如此，这仍是具有重要意义的一步，因为并没有什么原因表明我们不能用同样的方法来移动剩余的 63 个盘子。（实际上，我们确实必须用同样的方式来完成，因为必须再次移动其中的一个最大的盘子。）

＊＊＊分而治之

这正是递归的思想。我们已经描述过如何处理关键的步骤，并断言本质上可以用同样的方法来完成剩余的问题，这也是分而治之的思想：为解决一个问题，先将此任务分成越来越小的部分，其中的每个部分都比原始问题易于解决。

3. 细化

为了正式写出算法，我们需要知道每一步用哪个塔作为临时存储，因而我们将调用如下说明的函数：

```
void move(int count, int start, int finish, int temp);
```
前置条件：在塔 start 上至少有 count 个盘子。塔 temp 和 finish 上如果有盘子的话，每个都比塔 start 最上面的 count 个盘子中的任何一个都大。

后置条件：在 start 上最上面的 count 个盘子被移到 finish 上；temp(用作临时存储)返回到起始状态。

*** 停止规则

假设我们的任务将在有限步骤内结束(即使它意味着世界末日!)，那么必须有某种方法使递归停止。明显的停止规则是，当没有要移动的盘子时，就无事可做了。现在可以编写完整的程序来体现这些规则，主程序是：

```
const int disks=64;      //  Make this constant much smaller to run program.
void move(int count,int start,int finish,int temp);

/* Pre: None.
   Post: The simulation of the Towers of Hanoi has terminated. */
main()
{
  move(disks,1,3,2);
}
```

*** 递归函数

完成此任务的递归函数为：

```
void move(int count,int start,int finish,int temp)
{
  if (count>0) {
     move(count-1,start,temp,finish);
     cout <<"Move disk "<<count<<" from "<<start
          <<" to "<<finish <<"."<<endl;
     move(count-1,temp,finish,start);
  }
}
```

4. 程序跟踪

当递归函数应用于非常小的程序例子时，研究它的一个有用的工具是构造跟踪其活动的描绘。图 5.4 中所示的这种踪迹来源于盘数为 2 的汉诺塔，图中的每个框表示一次调用所发生的情况。最外层调用 move(2, 1, 3, 2)(主程序做出的调用)本质上导致下面 3 个语句的执行，如图中所示的最外部的框内的语句。

```
move(1, 1, 2, 3);              //  Move 1 disk from tower 1 to 2 (using tower 3).
cout <<"Move disk 2 from tower 1 to tower 3."<<endl;
```

```
move(1, 2, 3, 1);          //  Move 1 disk from tower 1 to 2 (using tower 1).
```

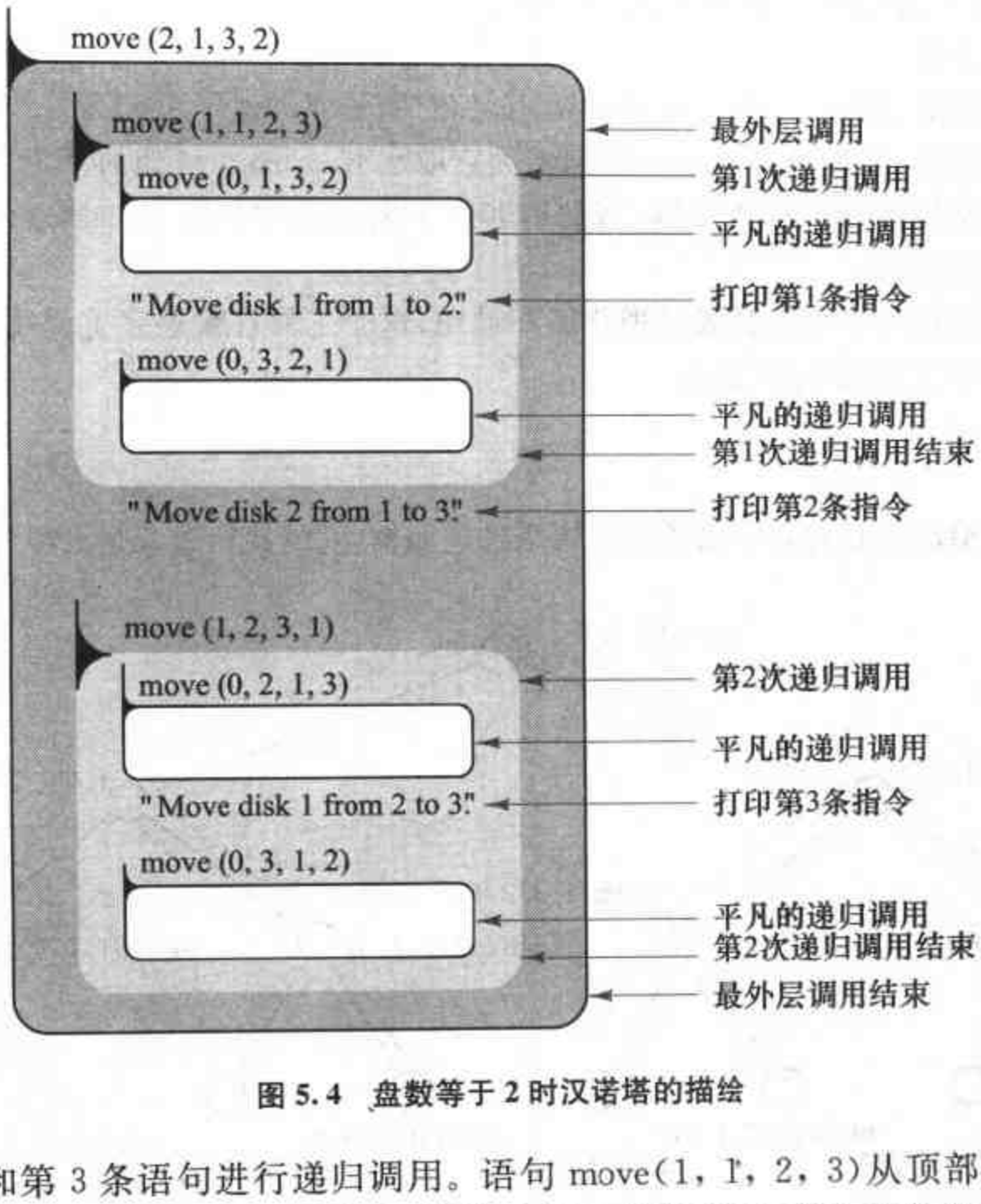


图 5.4　盘数等于 2 时汉诺塔的描绘

其中第 1 条和第 3 条语句进行递归调用。语句 move(1，1，2，3)从顶部再次启动函数 move，但现在却是用新的参数。因此这条语句本质上导致下面 3 条语句的执行，如图中所示的第一个内部框内的语句：

```
move(0, 1, 3, 2);          //  Move 0 disks.
cout << "Move disk 1 from tower 1 to tower 2." << endl;
move(0, 3, 2, 1);          //  Move 0 disks.
```

如果愿意的话，可以认为这 3 条语句写出来的目的是替代调用 move(1，1，2，3)，但因为它们构成了对此函数的一次新的不同的调用，所以认为它们与最外层调用的语句有不同的灰度。这些语句在图中用浅灰色打印显示。

对应于此调用的框的后面是输出语句，然后是对应调用 move(1，2，3，1)的第二个框。但在到达这些语句前，另有来自第一个内部框中的两次递归调用。也就是说，我们接下来必须展开调用 move(0，1，3，2)。但是当参数 count 为 0 时，函数 move 不做任何工作；因此这个调用 move(0，1，3，2)不执行更多的函数或其他语句。我们在图中将它对应于第一个空框。

在这个空调用后面是在第一个内部框中显示的输出，然后是不做任何事情的另一个调用，之后就完成了 move(1，1，2，3)的工作，并返回它被调用的位置。下面的语句则是在外

部框中的输出语句，最后语句 move(1，2，3，1)被完成。这个调用产生的语句显示在第二个内部框中，它们再依次展开为所示的更多的空框。

＊＊＊魔术师学徒

有了用我们的方法产生的所有递归调用，我们所研究的例子可能会导致你把递归比作魔术师学徒的寓言，当他施魔法让一把扫帚为它取水时，却不知道如何停止它，他只好将它劈成两半。于是扫帚开始自我复制，直到有非常多的扫帚为他取水，如果其主人不回来的话，灾难将随之发生。

现在我们使用另一个工具来可视化递归调用，这个工具在管理多重调用方面比程序跟踪更有效。这个工具就是递归树。

5. 分析

图 5.5 中给出的是有 3 个盘子的汉诺塔的递归树，它的执行过程是沿着浅灰色的路径。

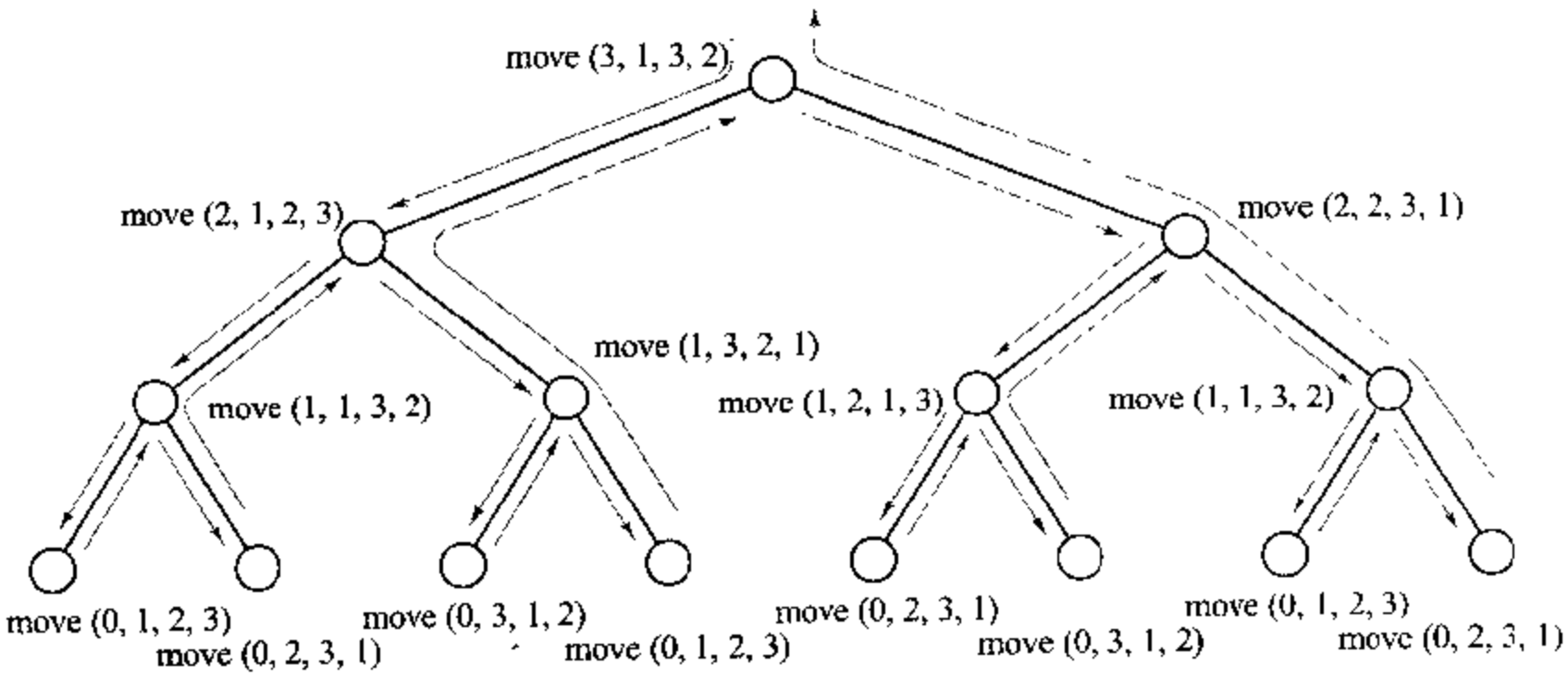


图 5.5　3 个盘子的递归树

注意我们的汉诺塔程序不仅产生了关于这个任务的一个完整的解答，而且产生的是最佳的解答。实际上，除了可能冗余的内含物和下面这些无用的指令序列，它是所能找到的惟一解答。

```
Move disk 1 from tower 1 to tower 2.
Move disk 1 from tower 2 to tower 3.
Move disk 1 from tower 3 to tower 1.
```

为了说明这种不可简化的解答的惟一性，注意在每个阶段，需要完成的任务可概括为将一定数目的盘子从一个塔移到另一个塔。没有其他的办法来完成这项任务，只能将除了底部以外的其他盘先移走，其中或许会做一些多余的移动。然后再移动底部的盘子，此时也可能会做更多的多余移动，最后再次移动上面的那些盘子。

＊＊＊递归的深度

接下来，我们来查明在返回并停止不干前，要进行多少次递归。函数 move 第一次被调用时，count ＝＝64，并且每次递归调用将 count 的值减 1。因此，如果我们排除不计 count ==0 时不做任何工作的调用的话，递归的合计深度为 64。即，如果我们准备画出此程

序的递归调用树的话，它在树叶以上会有 64 层。除了树叶，每个顶点导致两次递归调用（也写成一条指令），因此每一层的顶点数正好是其上层顶点数的两倍。

通过考虑它的递归调用树（即使它太大了而无法画出），可以简单地计算出移动 64 个盘子需要多少条指令。需要为树中除树叶（它以 count==0 调用）之外的每个顶点打印一条指令。非叶结点的数目是：

*** 移动的总次数

$$1+2+4+\cdots+2^{63}=2^0+2^1+2^2+\cdots+2^{63}=2^{64}-1。$$

因此 64 个盘子要求的移动次数总共是 $2^{64}-1$。我们可以近似估计一下这个数有多大：

$$10^3=1000\approx 1024=2^{10}$$

（这一简单的事实确实值得记住，在讨论计算机时经常用到它：缩写为 K，意义为 1024，如 512K。）因此移动的次数近似于 $2^{64}=2^4\times 2^{60}\approx 2^4\times 10^{18}=1.6\times 10^{19}$，一年约有 3.2×10^7 秒。假设指令用发狂的每秒 1 条的速度执行（神父有大量的实践）。则全部任务会花费 5×10^{11} 年。天文学家估计宇宙的年龄小于 200 亿（2×10^{10}）年，因此，根据这个故事，世界确实会持续存在很长一段时间——是它已存在时间的 25 倍。

*** 时间和空间

应该注意的是，没有计算机曾经运行了完全的汉诺塔程序，因为它会因时间不足而失败，而不是因为空间不足。所需要的空间只是保持对 64 次递归调用的跟踪，但所需的时间却是 2^{64} 次计算所要求的。

练习 5.1

E1. 考虑如下定义的函数 $f(n)$，这里 n 是一个非负整数：

$$f(n)=\begin{cases}0 & \text{若 } n=0\\ f(\frac{1}{2}n) & \text{若 } n \text{ 为偶数}, n>1\\ 1+f(n-1) & \text{若 } n \text{ 为奇数}, n>1\end{cases}$$

计算 n 为下列值时 $f(n)$ 的值：

(1) $n=1$　　(3) $n=3$　　(5) $n=100$

(2) $n=2$　　(4) $n=99$　　(6) $n=128$

E2. 考虑如下定义的函数 $f(n)$，这里 n 是一个非负整数：

$$f(n)=\begin{cases}n & n\leqslant 1\\ n+f(\frac{1}{2}n) & \text{若 } n \text{ 为偶数}, n>1\\ f(\frac{1}{2}(n+1))+f(\frac{1}{2}(n-1)) & \text{若 } n \text{ 为奇数}, n>1\end{cases}$$

对下列每个 n 的值，画出递归树并计算 $f(n)$ 的值。

(1) $n=1$　　(3) $n=3$　　(5) $n=5$

(2) $n=2$　　(4) $n=4$　　(6) $n=6$

程序设计项目 5.1

P1. 比较本节中递归的阶乘函数和用非递归方式实现的阶乘函数的运行时间[①]，其中非递归函数可用以下方法得到：将一个局部变量初始化为 1，并用一个循环来计算乘积 $n! = 1 \times 2 \times \cdots \times n$。为得到所需 CPU 时间的有意义的比较，可能需要在驱动程序中编写一个循环，它重复几百次同样的阶乘计算。如果你打算计算一个大数的阶乘，那么可能会发生整数溢出。为防止发生溢出，可以将 n 和函数值定义成 double 类型而非 int 类型。

P2. 证实 hanoi 程序的运行时间的增长近似于 2^n 的常数倍，这里 n 是要移动的盘数。为此，需要让 disks 成为一个变量，注释掉向用户输出信息的那些行，并用几个连续的 disks 值运行这个程序，如 10，11，…，15。从 disks 的一个值到其下一个值的 CPU 时间是如何变化的？

5.2 递归的原理

5.2.1 设计递归算法

递归是一种工具，它允许程序设计员将精力集中在算法的关键步骤上，而不必一开始就想着要将它同所有其他的步骤进行耦合。像通常的问题求解一样，最初采用的途径一般是考虑几个简单的例子，当更好地理解了这些例子以后，就尝试用公式表示出更为普遍适用的方法。设计递归算法的一些重要方面如下：

- 找出关键步骤。以自问的方式开始："如何能将这个问题划分成多个部分？"或者"如何完成中间的关键步骤？"确信你的答案简单却普遍适用。不要提出大量的特殊案例，并且这些特殊案例又仅适用于小问题或者是在大型程序的开头和结尾之处。一旦朝着解决方案迈出了简单的一小步，就要看看是否也能用同样的或类似的办法完成问题的其余部分。如果必要的话，就修改你的方法，使得它充分通用。
- 找出停止规则。停止规则象征着问题或问题的某个合适部分已被完成。停止规则通常是小的、无关紧要的或不需递归就能简单处理的特殊情况。
- 列出算法大纲。结合停止规则和关键步骤，并使用一个 if 语句在它们之间进行选择。现在能够编写出主程序和递归函数，递归函数描述如何执行关键步骤，直到适用停止规则。
- 检查终止。接下来很重要的一步是验证递归总会终止。从一种常规情况开始，检查在有限步骤内停止规则能被满足并且递归终止。同时确信算法正确地处理了特例。当任何算法被调用且什么也不做时，它们应当能够完美地返回。但特别重要的是，递归算法就应当是这样，因为什么也不做的调用通常就是停止规则。同样注意：什

① 需要标准的头文件 ctime.h 或 time.h，它访问函数包以计算 C 或 C++程序所使用的 CPU 时间。参见附录 C 中关于此软件包的更多细节。

么也不做的调用通常不是递归函数的错误，因此一般让递归函数在执行空调用时生成一条信息的做法并不恰当，而应该仅让它默默地返回。

- 画出递归树。分析递归算法的关键工具是递归树。我们已经从汉诺塔的例子中了解到，树的高度与程序要求的内存量密切相关，树的整个规模反映了执行关键步骤的时间，并因此反映了程序将使用的全部时间。一般画出适合于问题的一或两个简单例子的递归树会具有高度的启发性。

5.2.2 递归如何工作

＊＊＊设计与实现

应该在我们的思想中将递归如何在计算机中真正实现这一问题与在设计算法中使用递归的问题独立开来。

- 在设计阶段，应该使用证明合适的所有问题求解的方法，而递归是这些工具中最可行和功能最强大的一种。
- 在实现阶段，需要考虑几种算法中的哪一种在当前情况下是最佳的。

在计算机系统中至少有两种办法来实现递归。目前的第一种办法仅在某些大型系统中可用，但随着计算机设备成本和能力的不断变化，这种办法很快会更通用。我们考虑两种方法的要点在于，虽然确实需要考虑递归的空间和时间，但是对它们的考虑应该独立于算法设计的过程，因为将来不同的计算机设备会导致不同的能力和限制。

1. 多处理器：并发

或许考虑实现递归的最自然的办法是不要把每个函数看成占用同一计算机的不同部分，而是将每个函数看成运行在独立的机器上。那样，当一个函数调用另一个函数时，它启动了对应的机器运行。而当另一台机器完成其工作时，它会将答案送回给第一台机器，使得第一台机器然后能继续运行自己的任务。如果一个函数两次递归调用自己，则它只是启动了两台其他的机器来执行与它相同的指令。当这些机器完成了自己的工作后，会将答案送回给启动它们的机器。如果它们依次进行递归调用，则它们也只是启动更多的机器工作。

＊＊＊成本

过去，中央处理器是计算机系统中最昂贵的组件，一个计算机系统包含有多个处理器的想法曾经被认为是过分的奢侈。现在处理能力的价格与其他计算成本相比已显著下降，在不久的将来，我们很可能会看到大型计算机系统的组件中将包含几百个甚至是上千个同样的微处理器。当这种情况成为现实后，在多处理器上实现递归即使不是必然的，也会成为普遍的现象。

＊＊＊并行处理

＊＊＊并发

有了多个处理器，程序设计员应该不再将算法只考虑成操作的一个线性序列，而是应意识到算法的某些部分经常能与其他部分并行处理，同时进行的处理称为并发。目前，关于并发处理及其通信方法的研究是计算机科学研究中的一门活跃学科，它们的重要发展毫无疑

问地会改进以后几年算法描述和实现的方式。

2. 单处理器实现:存储区

为了判断如何能够在单处理器系统中有效地实现递归,我们暂时将递归放一放,先去考虑在简单的计算机中机器语言指令的原语层,调用一个函数需要哪些步骤。

* * *返回地址

* * *局部变量

任何计算机的硬件有一个受限范围的指令,它包含在指定存储字或 CPU 内称为寄存器的特殊位置上进行算术运算、在内存和寄存器之间来回移动数据和转移(跳转)到指定地址的指令(和其他的指令)。当调用程序转移到一个函数的开始处,发出调用的位置的地址必须被存储在内存中,否则此函数就无法记得返回到哪里。调用参数的地址或值也必须被存储,使得函数能找到它们,而且调用程序也能在此函数返回后依次找到答案。当函数启动时,它会在自己的局部变量和存储区上进行各种计算。然而,一旦此函数结束,这些局部变量将丢失,因为它们在函数外部是不可用的。当然此函数用了 CPU 中的寄存器来进行自己的计算,因此通常这些寄存器在函数完成后和调用前有不同的值。而除了函数自己的调用参数或全局变量(副作用)以外,按惯例不希望它再改变任何东西。因此通常函数会保存它将用到的所有寄存器,并在自己返回前恢复它们的值。

* * *存储区

总之,当一个函数被调用时,它必须有存储区(或许散布在几个区域),必须保存它将修改的寄存器或任何其他东西。它也使用存储区保存自己的返回地址、调用参数和局部变量。当它返回时,它不再需要局部存储区中的任何东西。

这样,我们通过改变存储区来实现函数调用,这一行动将代替我们以前考虑的改变处理器的做法。在这些考虑中,函数是否被递归调用实际上并无区别。在递归情况下,假如我们谨慎地把两次递归调用看成是不同的,因此不会将一个调用的存储区与另一个调用的存储区相混淆,至多只能算是在一个函数内部调用另一个函数时,混淆不同函数的存储区。对非递归函数,存储区可以是一个固定的区域,永久地保留,因为我们知道在一个函数调用返回后,才能发出另一个调用,而当第一个函数返回时,所存储的信息就不再需要。然而对递归函数,所存储的信息必须保留到外层的调用返回,因此内部的调用必须使用不同的区域作为临时存储。

注意,为非递归函数保留永久存储区的曾经的惯例实际上可能相当浪费,因为这样会消耗掉相当多的内存,而在此函数没有运行时,这些内存可以用作其他目的。尽管这样,在较老版本的语言如 FORTRAN 和 COBOL 中,分配给函数的存储还是采用这种方式,这也是为何这些老版本语言不允许递归的原因。

3. 重入程序

本质上,同样的多存储区问题也发生在另一种相当不同的环境中,即重入程序。在大型分时系统中,可能有许多用户并发地使用 C++编译器、文本编辑系统或数据库软件。这些系统程序都相当大,因此如果每个用户都复制一份,那么在内存中将同时保存完全相同的大量指令集的 30 或 40 个副本,这是对高速内存的大大浪费。通常代替这种做法的是在编写大

型系统程序如文本编辑器时，将指令放在一个区域，但将所有变量的地址或其他数据放在独立的区域。则在分时系统的内存中，将只有指令的一份副本，但对每个用户则各有一个独立的数据区域。

这种情况有点类似于学生在教室里参加考试，考题书写在黑板上，因此所有学生只会读到一个考题集合，但每个学生分别将答案写到不同的纸张上。不同的学生在同一时间读取同样的或不同的考题不存在困难，而有了不同的纸张，学生们的答案也不会互相混淆。见图 5.6。



图 5.6 并发、重入过程的例子

4. 数据结构：栈和树

＊＊＊栈

我们还必须说明用于跟踪函数的所有这些存储区的数据结构，为此让我们着眼于函数调用树。因要内部函数能访问在外部块中定义的变量，因要合适地返回到调用程序，所以必须在树中的每个点，记住从给定点返回根的路径上的所有顶点。当在树上移动时，会向路径的一端加入顶点或者从这一端删去顶点，路径的另一端（在根部）保持固定。因此路径上的顶点形成一个栈；函数的存储区同样地也可以保存为一个栈。此过程的图解见图 5.7。

＊＊＊时间和空间需求

从图 5.7 和我们的讨论，可以马上推断出实现递归（当然，它与目前使用的存储区数目有关）所需的空间量与递归树的高度直接成正比。没有仔细研究过递归的程序设计员有时错误地认为空间需求与树中的顶点总数有关。程序的时间需求与所完成的函数次数有关，因此与树中的顶点总数有关，但空间需求仅是从单个顶点反向回到根的路径上的那些存储区，因此树的高度反映了空间需求。一棵很均匀的、浓密的递归树意味着以较少的额外空间需求做大量工作的递归过程。

5.2.3 尾部递归（右递归）

＊＊＊废除栈元素

假设函数恰好在最后一个动作递归调用自己。我们知道，在递归的栈实现中，当启动递

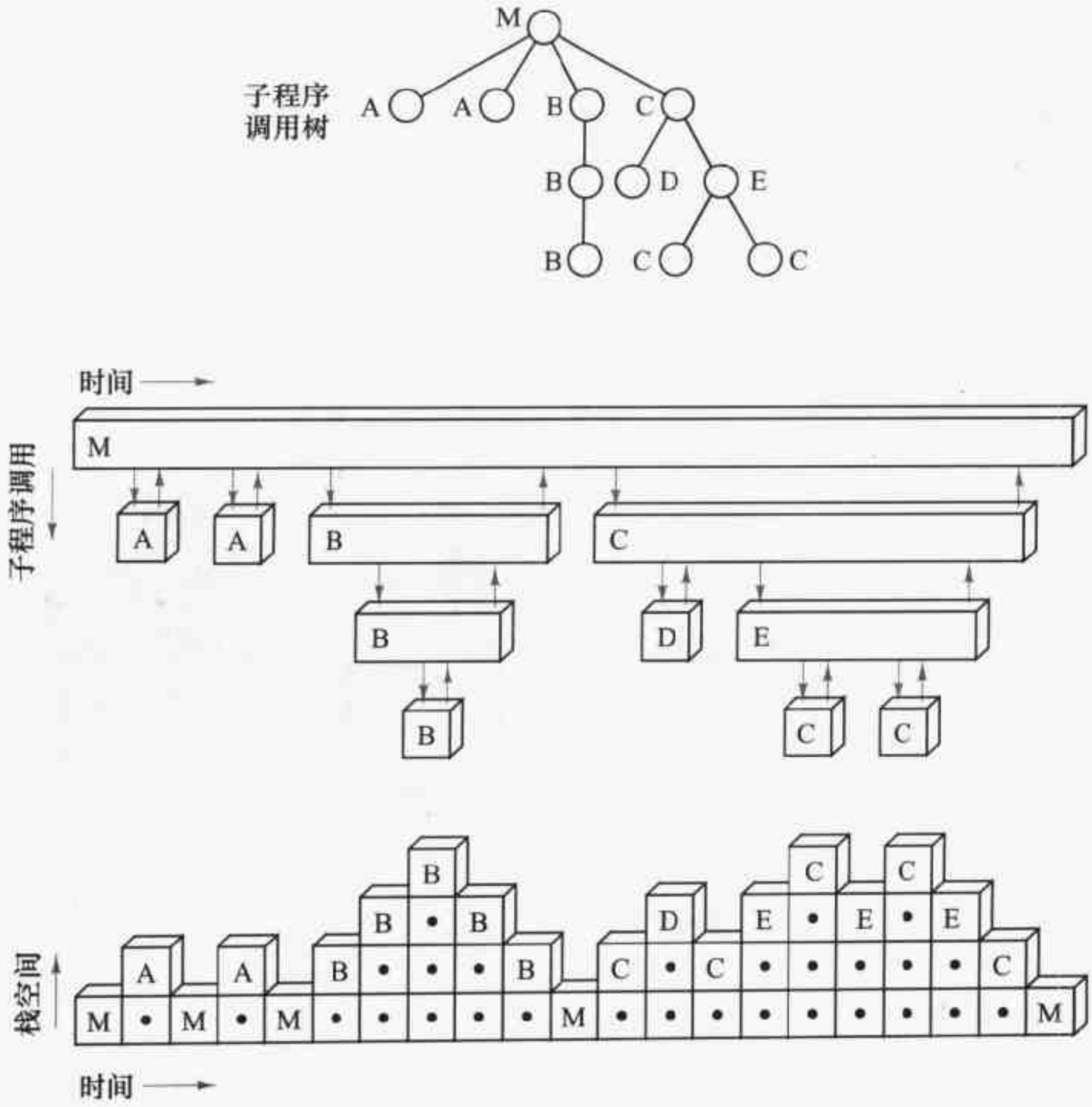


图 5.7 函数调用树和相关的栈帧

归调用时，会将函数的局部变量压入栈中。当递归调用终止时，将从栈中弹出这些变量并由此恢复它们先前的值。但因为现在递归调用是函数的最后一个动作，因此做这一步没有什么意义，此函数马上终止并且会立即丢掉刚刚恢复的局部变量。

因此，当函数恰好是在最后一个动作递归调用自己时，就没有必要使用栈，就如我们所看到的，因为不必保留任何局部变量。所需做的事情就是将哑调用参数设置成它们的新值（就像为内部递归调用指定的那样）并转移到函数的开始处。我们概括这条原理以供将来参考。

如果函数最后执行的语句是对函数自身的递归调用，则可以通过将调用参数赋给递归调用中指定的值并重复整个函数而消除此调用。

这种变换的过程如图 5.8 所示。(a)部分显示了调用程序 M 和递归函数 P 的数个副本所使用的存储区，每个 P 都由它的前一个函数进行调用。浅灰色的箭头显示了从一个函数调用到下一个函数调用的控制流，块显示了系统所维护的存储区。由于 P 对自己的每次调用都是最后一个动作，于是在从调用返回后没有必要维护存储区。减少后的存储区如(b)部

分所示。最后,(c)部分显示了在图的同一层以迭代方式重复调用 P。

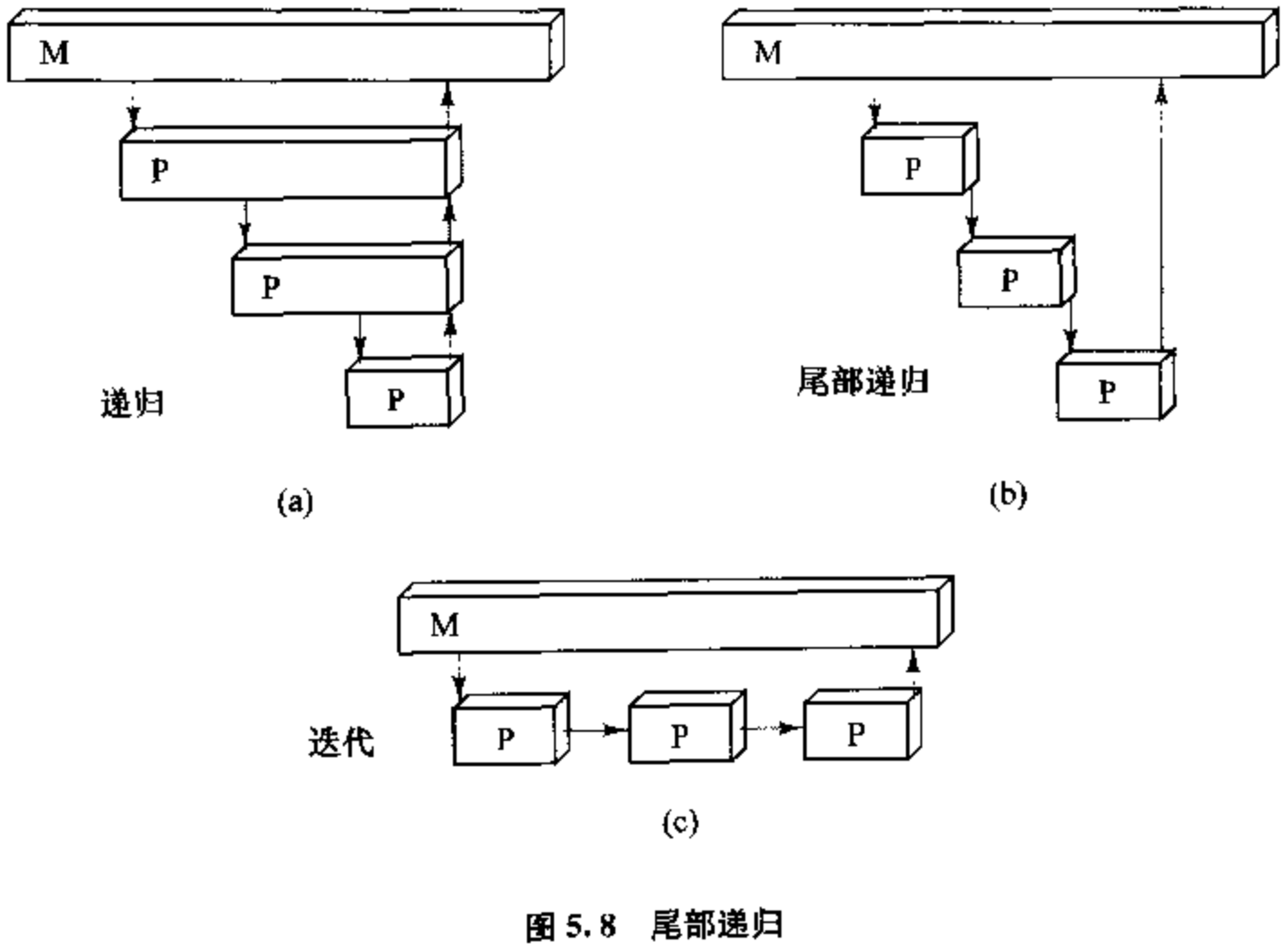


图 5.8 尾部递归

* * * 尾部递归

函数的最后执行语句是一个递归调用这样的特殊情况尤其重要,因为这种情况经常发生,我们称之为尾部递归(亦可称为右递归)。需要注意的是,尾部递归的意思是最后执行的语句是一个递归调用,而不是此递归调用必须是函数中出现的最后一个语句。例如,尾部递归可以是 swtich 语句或 if 语句中的分句,在其后还会出现其他的程序行。

* * * 时间和空间

对大多数编译器来说,在程序中留下或移去尾部递归,在执行时间上不会有多少差别。然而,如果空间因素重要的话,则通常应移去尾部递归。如果必要的话,可以重新安排终止条件,通常可以用一个 do while 或 while 语句来重复这个函数。

例如,考虑像汉诺塔这样的分而治之的算法。在函数 move 中的第二个递归调用是尾部递归,但第一个不是。通过移去尾部递归,函数 move 的原始递归程序可以表达为

* * * 不带尾部递归的汉诺塔

```
void move(int count,int start,int finish,int temp)
/* move: iterative version
   Pre:  Disk count is a validdisk tobe moved.
   Post: Moves count disks from start to finish using temp for temporary storage. */
{
   int swap;                    //  temporary storage to swap towers
   while (count> 0) {           //  Replace the if statement with a loop.
    move(count-1,start,temp,finish); //  first recursive call
    cout << "Move disk "<< count << " from "<< start
         << " to "<< finish << "."<< endl;
    count--;                    //  Change parameters to mimic the second recursive call.
    swap=start;
```

```
    start=temp;
    temp=swap;
  }
}
```

如果一开始就能看出这个问题，那么我们就显得非常聪明。但是现在我们通过其他因素发现它，也可以得到一种自然的解释。把两个塔 start 和 temp 看作是在同一类中：当我们慢慢地将所有盘子移到 finish 上时，想用这两个塔作为临时存储。为了将一堆盘子移到 finish 上，则首先必须将除了底部盘子以外的所有盘子都移到 start 和 temp 之一上，然后再将底部的盘子移到 finish 上，并在交换 start 和 temp 后重复，继续在 start 和 temp 之间移动除底部的盘子以外的所有盘子，然后在每一遍，都会将一个新的底部盘子移到 finish 上。

5.2.4 不用递归的时机

1. 阶乘

考虑下面两个计算阶乘的函数。我们已经了解了用递归实现的这个函数：

```
int factorial(int n)
/* factorial: recursive version
   Pre:  n is a nonnegative integer.
   Post: Return the value of the factorial of n. */
{
   if (n==0) return 1;
   else      return n* factorial(n-1);
}
```

这里有一个几乎同样简单的迭代版：

```
int factorial(int n)
/* factorial: iterative version
   Pre:  n is a nonnegative integer.
   Post: Return the value of the factorial of n. */
{
   int count, product=1;
   for(count=1;count<=n;count++)
      product*=count;
   return product;
}
```

这两个程序中哪一个使用的存储空间更少？乍一看好像是用递归实现的那一个，因为它没有局部变量，而以迭代实现的程序有两个局部变量。但实际上（见图 5.9），递归程序会创建一个栈并在其中放入 n 个数：$n, n-1, n-2, \cdots, 2, 1$。这是每次递归前它的调用参数，并且当它从递归退出时，会像第二个程序那样用同样的顺序将这些数相乘。将 $n = 5$ 应用于递归函数时的执行过程如下：

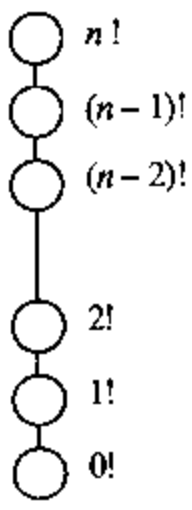


图 5.9 计算阶乘的递归树

$$
\begin{aligned}
\text{factorial}(5) &= 5 * \text{factorial}(4) \\
&= 5 * (4 * \text{factorial}(3)) \\
&= 5 * (4 * (3 * \text{factorial}(2))) \\
&= 5 * (4 * (3 * (2 * (\text{factorial}(1)))))
\end{aligned}
$$

$=5*(4*(3*(2*(1*\text{factorial}(0)))))$

$=5*(4*(3*(2*(1*1))))$

$=5*(4*(3*(2*1)))$

$=5*(4*(3*2))$

$=5*(4*6)$

$=5*24$

$=120$

因此递归程序比迭代版本占用更多的存储空间，并且它也会花费更多的时间，因为它必须存储和得到所有的数并且将它们相乘。

2. 斐波纳契数

比阶乘更浪费的例子（在某些课本中同样也是作为一个明显推荐的程序出现）是斐波纳契数的计算，它通过下面的递推关系定义：$F_0=0, F_1=1, F_n=F_{n-1}+F_{n-2}$ for $n\geqslant 2$。

它的递归程序严格遵从此定义：

```
int fibonacci(int n)
/* fibonacci: recursive version
   Pre:  The parameter n is a nonnegative integer.
   Post: The function returns the nth Fibonacci number. */
{
   if (n<=0)       return 0;
   else if(n==1)   return 1;
   else            return fibonacci(n-1)+fibonacci(n-2);
}
```

实际上，此程序相当漂亮，因为它属于分而治之的形式：通过计算两个更小的问题来得到答案。但是就像我们所看到的，在这个例子中，它不是“分而且治之”，而是“分而且复杂”。

为了评价这个算法，作为一个例子，我们考虑 F_7 的计算，它的递归树如图 5.10 所示。函数首先必须得到 F_6 和 F_5。得到 F_5 需要 F_5 和 F_4，以此类推。但在为 F_6 计算 F_5 后，会将 F_5 丢掉，使得 F_5 在接下来为计算 F_7 时不再可用。因此如递归树所示，递归程序不必要地再三重复同样的计算，进一步的分析将留作练习。结果用于计算 F_n 的递归函数所用的时间量以 n 的指数级增长。

同阶乘函数一样，我们能给出一个简单的迭代程序，注意从 0 开始并且仅保存 3 个变量：当前的斐波纳契数和它的前两个前趋。

```
int fibonacci(int n)
/* fibonacci: iterative version
   Pre:  The parameter n is a nonnegative integer.
   Post: The function returns the n th Fibonacci number.*/
{
   int last_but_one;        //  second previous Fibonacci number, F_{i-2}
   int last_value;          //  previous Fibonacci number, F_{i-1}
   int current;             //  current Fibonacci number F_i
   if(n<=0)return 0;
   else if(n==1)return 1;
   else {
```

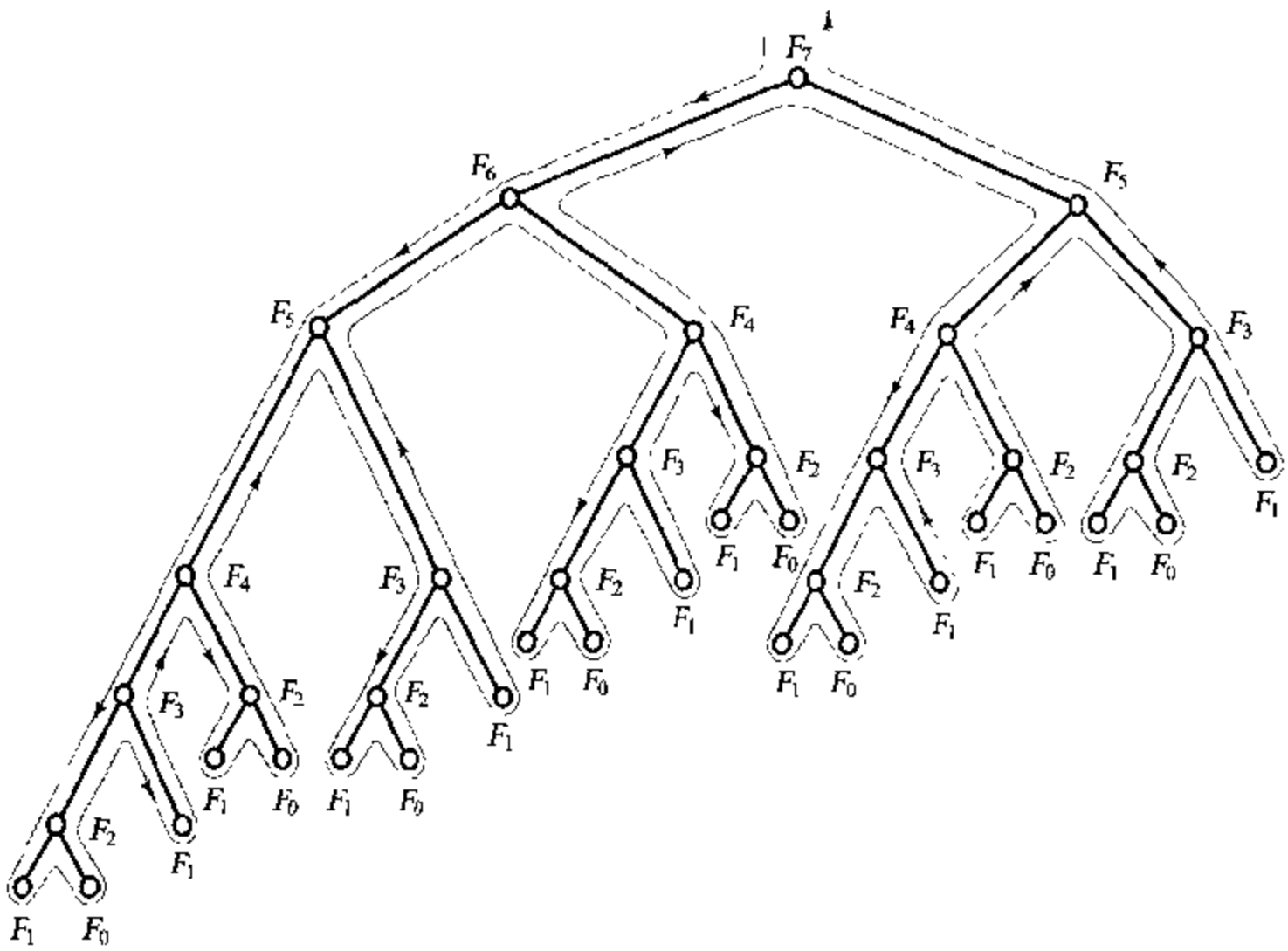


图 5.10　计算 F_7 的递归树

```
    last_but_one=0;
    last_value=1;
    for (int i=2; i<=n; i++) {
      current=last_but_one+last_value;
      last_but_one=last_value;
      last_value=current;
    }
    return current;
  }
}
```

这个迭代函数所用的时间显然是按 n 成线性增长(即直接正比于 n),因此这个函数的时间与递归函数的指数级时间的差异是巨大的。

3. 递归和迭代的比较

后面的这个例子与完全使用递归的例子之间的本质区别是什么？为回答这个问题,我们再次转向检查递归树。已经清楚的是,对递归树的研究将提供大量有用的信息来帮助我们决定何时应该或不应该使用递归。

* * * 链

如果函数仅对自身进行一次递归调用,则它的递归树形式就非常简单:它是一条链,即每个顶点仅有一个子结点。这个子结点对应于发生的单个递归调用,我们很容易理解这样简单的树。对阶乘函数,它只是从 $(n-1)!$ 直到 $1!$ 的一串计算阶乘的请求。如果从下往上而不是从上往下读取这棵递归树,则可以从递归程序立即得到迭代程序。当将树简化成一条

链时，则从递归到迭代的变换就非常简单，迭代函数可能同时节省空间和时间。

注意函数对自己仅做一次递归调用与函数中仅有一个位置做递归调用完全不是一回事，因为这个位置可能是在一个循环中，也可能有两个地方发出递归调用（如同时在 if 语句的两个分支中）但只有一个调用会真正发生。

*** 多重任务

计算斐波纳契数的递归树不是一个链，而是包含许多象征多重任务的顶点。当函数运行时，它会在遍历这棵树时创建一个栈来使用。但如果存在栈中的结果被废弃，而不是保存在某个其他的数据结构中供将来使用，则会发生大量的重复工作，就像斐波纳契数的递归计算一样。

*** 修改数据结构

在这样的情况下，用另一种数据结构来代替栈更为可取，这种数据结构允许到位置而不是到顶部的引用。对斐波纳契数，我们仅需要两个额外的临时变量来保存计算当前数所需要的信息。

*** 递归的消除

最后，通过创建一个显式栈，可能将任何递归程序改编成非递归的形式。但是结果形成的程序几乎总是比递归版本更为复杂且难以理解。转化程序以消除递归的惟一原因是被迫采用一门不支持递归的语言进行程序设计，而用这样的语言编写程序的可能性将越来越小。

4. 斐波纳契数和汉诺塔的比较：输出规模

斐波纳契数的递归函数和汉诺塔的递归过程有非常类似的分而治之形式。它们本质上都由两个对自身的递归调用组成，它们都是比原来稍小的问题。那么为何汉诺塔程序效率非常高，而斐波纳契数程序的效率又十分低下呢？分析输出的规模可以得到答案。我们在 Fibonacci 中仅计算一个数，并且希望几步就完成计算，就像迭代函数所做的，但递归函数没有做到。另一方面，Hanoi 的输出规模是要打印的指令数，它随盘子数呈指数级增长。因此汉诺塔的任何过程必然要求时间随盘子数呈指数级增长。

5.2.5 指导原则和结论

因而，在决定是否用递归或非递归形式编写特定算法时，一个好的出发点是考虑递归树。如果它形式简单，则迭代版可能更好。如果它包含多重任务，则不同于栈的数据结构将更合适，且不存在递归的需要。如果递归树显得相当茂密，几乎没有多重任务，则递归可能是自然的选择。

*** 自顶向下的设计

用于解决递归的栈可以看作是此程序延缓的任务的列表。如果事先能轻易地构造出这个表，则迭代可能更好；如果不行，则可能选择递归。递归有几分像是自顶向下的问题求解方法，它将问题分成块或者选出一个关键步骤并延缓其他步骤。迭代更多地是自底向上的方法，以所知的开始，并以此逐步构造出解答。

*** 栈或递归

递归能被迭代和栈所置换，这一点是事实。反之（证明见参考书目），任何使用栈的（迭

代的)程序能被没有栈的递归程序替换,这一点也是事实。因此细心的程序设计员不仅要探究是否应消除递归,而且要问程序何时包括栈,是否递归的引入能得到更自然和更易理解的程序,这个程序能否导致方法和结果的改进。

练习 5.2

E1. 在 F_n 的递归计算中,准确判断每个较小的斐波纳契数将被计算多少次。由此判断这个递归函数时间和空间需求的数量级。(或者通过建立并求解一个递归关系——自顶向下的方法,或者通过找到简单案例的答案并用数学归纳法证明它更通用——自底向上的方法。)

E2. 两个正整数的最大公约数(gcd)是能同时整除它们的最大整数。如此,例如 8 和 12 的 gcd 是 4,9 和 18 的 gcd 是 9,16 和 25 的 gcd 是 1。

(1)编写一个非递归函数 int gcd(int x, int y),这里要求 x 和 y 是正整数,此函数搜寻正整数,直到发现同时整除 x 和 y 的最大整数。

(2)编写一个递归函数 int gcd(int x, int y)实现 Euclid 算法:若 y=0,则 x 和 y 的 gcd 是 x;否则 x 和 y 的 gcd 等于 y 和 x % y 的 gcd①

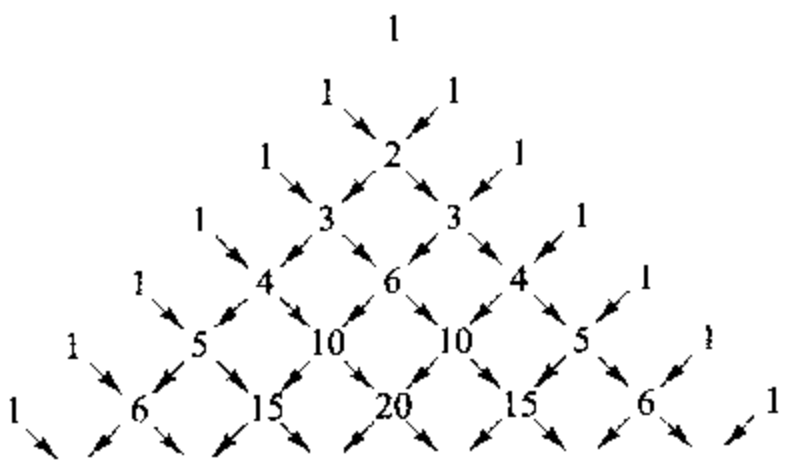
(3)将(2)部分的函数改写成迭代形式。

(4)讨论每种方法的优缺点。

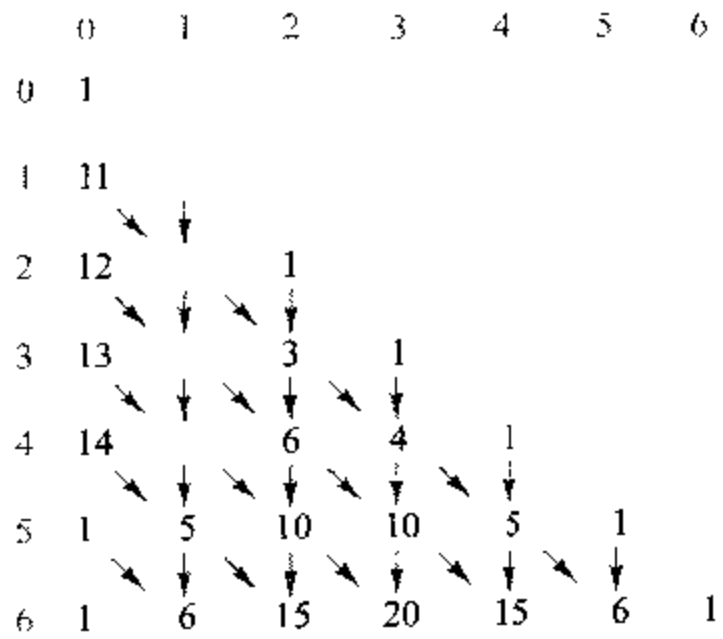
E3. 二项式系数可用下列递归关系定义,这是 Pascal 三角形的思想。Pascal 三角形的顶如图 5.11 所示。

对 $n \geqslant 0, C(n, 0)=1$ 且 $C(n, n)=1$

对 $n>k>0, C(n, k)=C(n-1, k)+C(n-1, k-1)$



(a) 对称形式



(b) 在正方形数组中

图 5.11 二项式系数的 Pascal 三角形的顶部

(1)编写一个递归函数用上述公式产生 $C(n, k)$。

① 记住% 是模运算符:x% y 的结果是由整数 x 除以非零整数 y 后的余数。

(2)画出计算 $C(6, 4)$的递归树。

(3)用一个矩形数组,n 表示行,k 表示列,编写一个非递归的程序产生数组左下半部中的 Pascal 三角形,即 $k \leqslant n$ 的那些元素。

(4)编写一个非递归的程序,既不使用数组也不使用栈,对任意的 $n \geqslant k \geqslant 0$,计算 $C(n, k)$。

(5)判断(1)、(3)和(4)中设计的每个算法近似的空间和时间需求。

E4. Ackermann 函数如下定义,它是一种判断在计算机上递归程序实现质量的标准手段。

$A(0, n)=n+1$ 对 $n \geqslant 0$

$A(m, 0)=A(m-1, 1)$ 对 $m>0$

$A(m, n)=A(m-1, A(m, n-1))$ 对 $m>0$ 且 $n>0$

(1)编写一个递归函数计算 Ackermann 函数。

(2)计算下列值。如果不可能得到这些值的任何一个,解释原因。

$A(0, 0)$ $A(0, 9)$ $A(1, 8)$ $A(2, 2)$ $A(2, 0)$

$A(2, 3)$ $A(3, 2)$ $A(4, 2)$ $A(4, 3)$ $A(4, 0)$

(3)编写一个非递归函数计算 Ackermann 函数。

5.3 回溯法:延缓工作

作为递归更加复杂的一种应用,考虑一个有名的难题,即如何在棋盘上放入 8 个皇后而让它们不能互相吃掉。在国际象棋的规则中,皇后能够吃掉与自己位于同一行、同一列或同一对角线(任一方向)上的其他棋子。棋盘有 8 行 8 列。

如何求解这个难题,答案决不明显,即使是伟大的 C. F. Gauss(高斯)在 1850 年考虑它时也没有得到全部解答。它似乎是不适合完全通过分析来解决的典型难题,而是要求运气,再加上试验和错误,或者是大量的彻底的(穷尽)计算。为了让你确信这个问题确实存在解答,我们在图 5.12 中给出了其中两个。

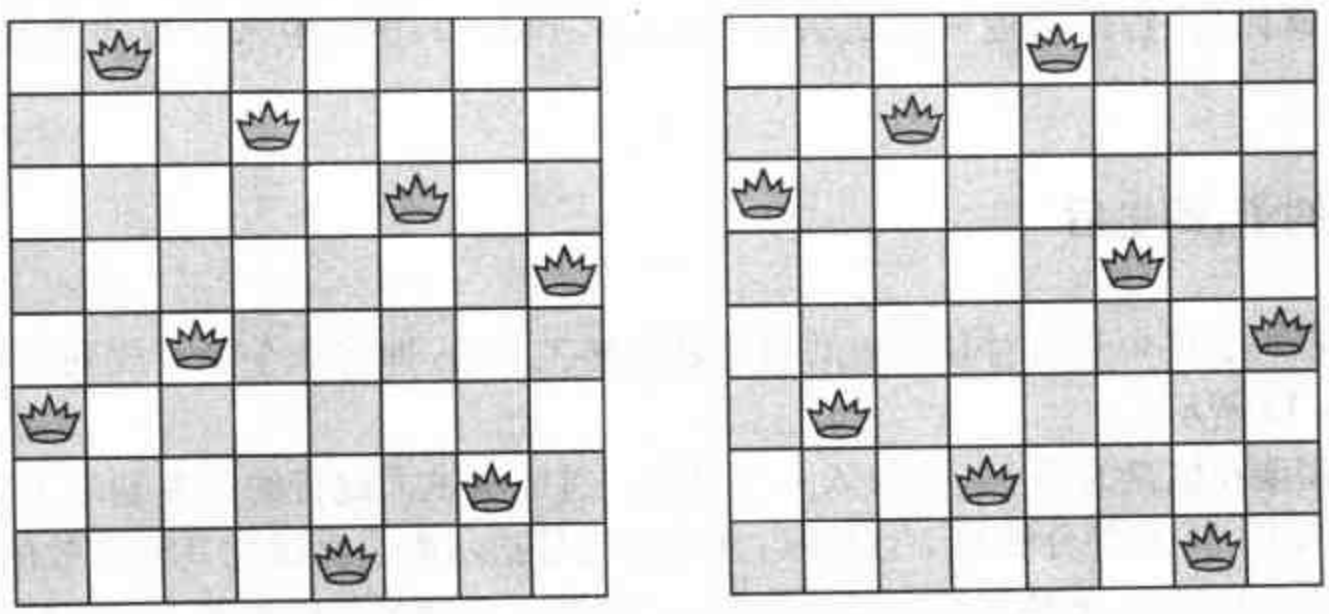


图 5.12 展示 8 个非攻击性皇后的两种配置

本节将设计两个程序来求解“八皇后”问题,它将说明数据结构的选择如何影响递归程序。

5.3.1 求解八皇后难题

试图求解八皇后问题的人通常会很快放弃靠小聪明找出所有(甚至是其中一个)解答的尝试,而会开始在棋盘上摆皇后,或者随机地或者可能以某种逻辑来摆放,但始终确信每个皇后都不能吃掉棋盘上已有的其他皇后。如果这个人足够幸运,他就能以这种方式进行下去并将8个皇后都摆在棋盘中,得到一种解答。如果不行的话,则必须拿走一个或多个皇后并将它们放在其他位置以继续搜寻解答。为了开始设计程序,我们先略述一下这种技巧。我们将它看成一个递归函数,它从棋盘上给定的皇后配置开始,查找所有的解答。我们称这个函数为 solve_from。现在设想使用一个名为 Queens 的类来表达皇后的部分配置(configuration),因此可以将 Queens configuration 作为参数传给递归函数 solve_from。在主程序中初次调用 solve_from 时,参数 Queens configuration 是空的。

```
* * * 概要
solve_from(Queens configuration)
  if Queens configuration 已包含 8 个皇后
    打印 configuration
  else
      对configuration 未设防的每个棋盘格子 p {
        在 configuration 的格子 p 增加 1 个皇后;
        solve_from(configuration);
        将皇后从 configuration 的格子 p 移走;
      }
```

这个框架说明使用递归意味着“继续到下一阶段并重复此任务。”在格子 p 中放入一个皇后只是试探性的,只有当我们能继续增加到8个皇后时才将它留在那里。无论是否能达到8个皇后,此过程都会返回,或者当它发现自己结束了,或者没有进一步的试探可能。在内部调用返回后,程序会返回去试探向 Queens configuration 中未设防的其他格子加入皇后。

5.3.2 例子:四皇后

先来看一下这个算法对较简单的问题是怎样工作的,即将4个皇后放到4×4的棋盘上,如图5.13所示。

我们需要在棋盘的每一行中放入一个皇后。首先尝试向此行的最左端放入皇后,这种选择如图5.13中(a)部分的第1行所示,其中的问号指示尚未尝试的其他有效的选择。在我们尝试这些选择前,先移到第2行并插入一个皇后。前两列都已被第1行中的皇后设防,在图中用删去的格子显示。第3,第4两列是空闲的,因此我们首先将皇后放在第3列并将第4列标上一个问号。接下来移至第3行,但此时所有的4个格子都被前面两行中的皇后设防,于是到达一个僵局。

当到达僵局时,必须回溯,即返回最近做出的选择并尝试另一种可能。这种情况如图

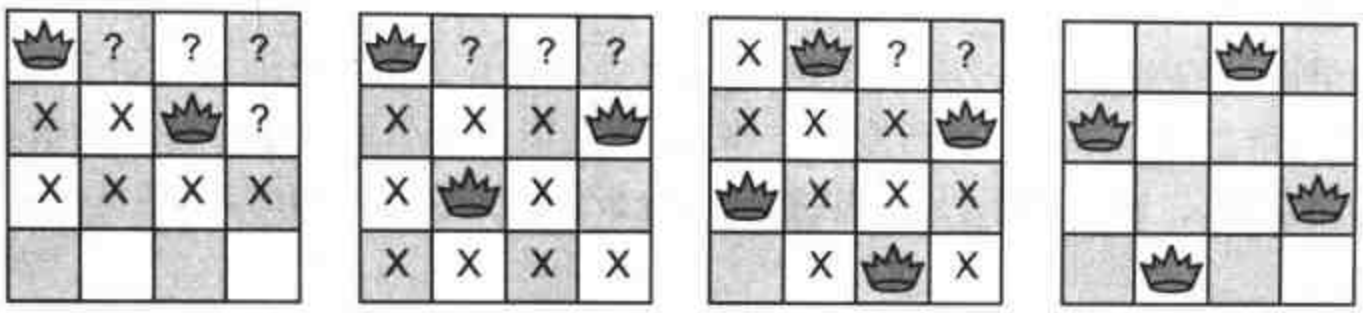


图 5.13 四皇后问题的解答

5.13 中(b)部分所示,它显示第 1 行中的皇后未改变,但第 2 行中的皇后被移到第二种可能位置(前次占据的位置因不再可能而被删去)。现在发现第 2 列是在第 3 行中放入皇后的惟一可能的位置,但在第 4 行所有的四个位置都被设防。因此再次到达不能再加入其他皇后的一个点,因而必须回溯。

在此处,对第 2 行已不再有其他选择,因此必须一直返回到第一行并将皇后放在下一个可能的位置即第 2 列,这种情况如图 5.13 中的(c)部分所示。现在发现在第 2 行中只有第 4 列是未设防的,因此必须在那里放上皇后。那么在第 3 行,第 1 列是惟一的可能,同时在第 4 行,仅有第 3 列是可能的。而这种皇后的放置却得到同样的 4×4 棋盘上四个非攻击性皇后问题的一种解答。

如果愿意找出所有的解答,可以继续使用同样的办法,回溯到最后所做的选择并将皇后移到下一个可能的位置。在(c)部分,第 4,第 3 或第 2 行已没有选择,因此现在返回到第 1 行,并将皇后放到第 3 列。这种选择导致(d)部分所示的惟一解答。

最后,我们尝试在第 1 行第 4 列放入皇后的可能性,但与(a)部分一样,这种情况没有解答。实际上,在第 1 行的第 3 列或第 4 列有皇后的配置正好分别是第 2 列或第 1 列有皇后的配置的镜像。如果对(c)部分所示的棋盘做一个左右反射,可以得到(d)中所示的棋盘,而在第 1 行第 4 列有皇后的棋盘,正好是在(a)和(b)部分中所示的棋盘的反射。

5.3.3 回溯

这种方法是所谓的回溯类算法中的典型,它试图通过构造部分解(partial solution),并总是保证部分解与问题的需求保持一致,从而完成关于问题的解的搜索。这个算法然后将部分解扩展到完善,但当发生与问题需求不一致的情况时,算法通过移去最近构造的解部分并进行倒退(回溯)以尝试另一种可能性。

回溯对以下情况证明是有用的:即最初会出现许多种可能性,但在进一步的测试中只有少数几种会继续存在。例如在调度问题中(如安排体育比赛),可能开始几场比赛容易安排,但要安排更多的比赛时,约束条件会显著地减少可能性的数目。或者用编译器设计问题作为例子,在某些语言中,直到几乎已读完一条语句时才能判断该语句的意义。例如考虑下面这对 Fortran 语句:

```
DO 17 K=1,6
DO 17 K=1.6
```

* * * 语法分析

这两个语句都是合法的：第一个语句启动一个循环，第二个语句将数 1.6 赋给名为 DO17K 的变量。(Fortran 忽略所有的空格，即便是标识符中的空格。)在这种情况下，不能马上推断出其意义，在对程序的文本进行语法分析(即分裂成译码)时，回溯是有用的方法。

5.3.4 总体大纲

1. 主程序

虽然对用于表达棋盘上皇后的位置的数据结构仍有许多细节需要补充，但我们已可以提供一个主程序来驱动已经列出概要的递归方法。

程序首先打印出关于程序功能的信息。由于通过解决更小的问题如四皇后问题有益于测试程序，因此在程序中允许用户指定所用的皇后数(称为 board_size)。程序中使用一个全局常量 max_board(定义在 queens.h 头文件中)来限制程序尝试放置的最大皇后数。

```
int main( )
/* Pre:  The user enters a valid board size.
   Post: All solutions to the n-queens puzzle for the selected board size are printed.
   Uses: The class Queens and the recursive function solve_from.*/
{
   int board_size;
   print_information( );
   cout<< "What is the size of the board?"<< flush;
   cin >>board_size;
   if (board_size<0||board_size>max_board)
      cout<< "The number must be between 0 and"<< max_board<< endl;
   else {
     Queens configuration(board_size);  //  Initialize empty configuration.
     solve_from(configuration);  //  Find all solutions extending configuration.
   }
}
```

2. Queens 类

变量定义 Queens configuration(board_size)使用了构造函数，它带有一个参数，Queens 类用这个参数设置用户选择的棋盘大小并初始化空的名为 configuration 的 Queens 对象。这个空的配置将作为参数传递给在棋盘上摆放皇后的递归函数。

第 5.3.1 节的概要显示 Queens 类需要以下这些方法：打印一个配置，在棋盘上某特定格子处向配置中增加一个皇后，移去这个皇后，测试某特定格子是否未被配置设防。另外，要想设计出函数 solve_from，还需要 Queens 的数据成员 board_size(保存棋盘的大小)和 count(保存已插入的皇后数)。

* * * 鸽巢原理

在创建了配置后，如何找出下一次要尝试的格子？一旦在给定行放入一个皇后，则由于这一行已被这个皇后设防，因此不会有人浪费时间再次搜索同一行，以期找到另一个位置来摆放另一个皇后，因为在每一行决不会有多于一个皇后。但是我们的目标是将 board_size 个皇后摆放到棋盘上，而棋盘上只有 board_size 行，随之我们得出：在每一行必定有且仅有

一个皇后。(这称为鸽巢原理:如果有 n 只鸽子和 n 个鸽巢,并且在同一个巢中不能多于一只鸽子,那么在每个巢中必定有一只鸽子。)因此,每次可以将皇后摆在棋盘的一行,从第 0 行开始,可以用单个数据成员 count 得知已到达哪一行。因此 count 不仅给出了配置中迄今为止的皇后总数,同时也给出了首个空行的下标。由此,我们总是尝试将下一个皇后插入到 Queens 配置的第 count 行。

Queens 类的主要方法的说明如下:

```
bool Queens::unguarded(int col) const;
```
后置条件:根据首个空行(第 count 行)第 col 列中的格子是否未被任何皇后设防而返回 true 或 false。

```
void Queens::insert(int col);
```
前置条件:在首个空行(第 count 行)第 col 列中的格子未被任何皇后设防。
后置条件:在第 count 行第 col 列的格子中插入一个皇后;count 加 1。

```
void Queens::remove(int col);
```
前置条件:在第 count-1 行第 col 列的格子中有一个皇后。
后置条件:移去上述皇后;count 减 1。

```
bool Queens::is_sovled() const;
```
后置条件:若已摆放的皇后数已经等于 board_size 则返回 true;否则返回 false。

3. 回溯函数 solve_from

有了上述这些决定,现在已能够编写在棋盘上摆放皇后的递归函数第 1 版的 C++代码。注意通过引用传递函数参数是为了节省用于复制 Queens 对象的时间。遗憾的是,虽然这个参数仅是一个输入参数,但在函数中先对它做了修改,然后又取消了修改,因此不能将它作为一个常量引用传递。

```
void solve_from(Queens &configuration)
/* Pre:  The Queens configuration represents a partially completed arrangement of
         nonattacking queens on a chessboard.
   Post: All n-queens solutions that extend the given configuration are printed. The
         configuration is restored to its initial state.
   Uses: The class Queens and the function solve_from, recursively.*/
{
   if (configuration.is_solved( ))configuration.print( );
   else
     for (int col=0; col<configuration.board_size; col++)
       if (configuration.unguarded(col)){
          configuration.insert(col);
          solve_from(configuration); //  Recursively continue to add queens.
```

```
        configuration.remove(col);
      }
}
```

5.3.5 细化:首选的数据结构及其方法

表示 Queens 配置的一种明显的办法是将棋盘作为正方形数组存储,数组的元素指示已摆放皇后的位置。这样的数组是我们首选的数据结构。在头文件中对它的表达是这样的:

```
const int max_board=30;
class Queens {
public:
  Queens(int size);
  bool is_solved() const;
  void print() const;
  bool unguarded(int col) const;
  void insert(int col);
  void remove(int col);
  int board_size;        //  dimension of board=maximum number of queens
private:
   int count;            //  current number of queens=first unoccupied row
   bool queen_square[max_board][max_board];
};
```

采用这个数据结构,加入新皇后的方法就很简单了:

```
void Queens::insert(int col)
/* Pre:  The square in the first unoccupied row (row count) and column col is not guarded
         by any queen.
   Post: A queen has been inserted into the square at row count and column col; count has
         been incremented by 1. */
{
  queen_square[count++][col]=true;
}
```

方法 is_sovled,remove 和 print 也都十分简单,因此将它们留作练习。

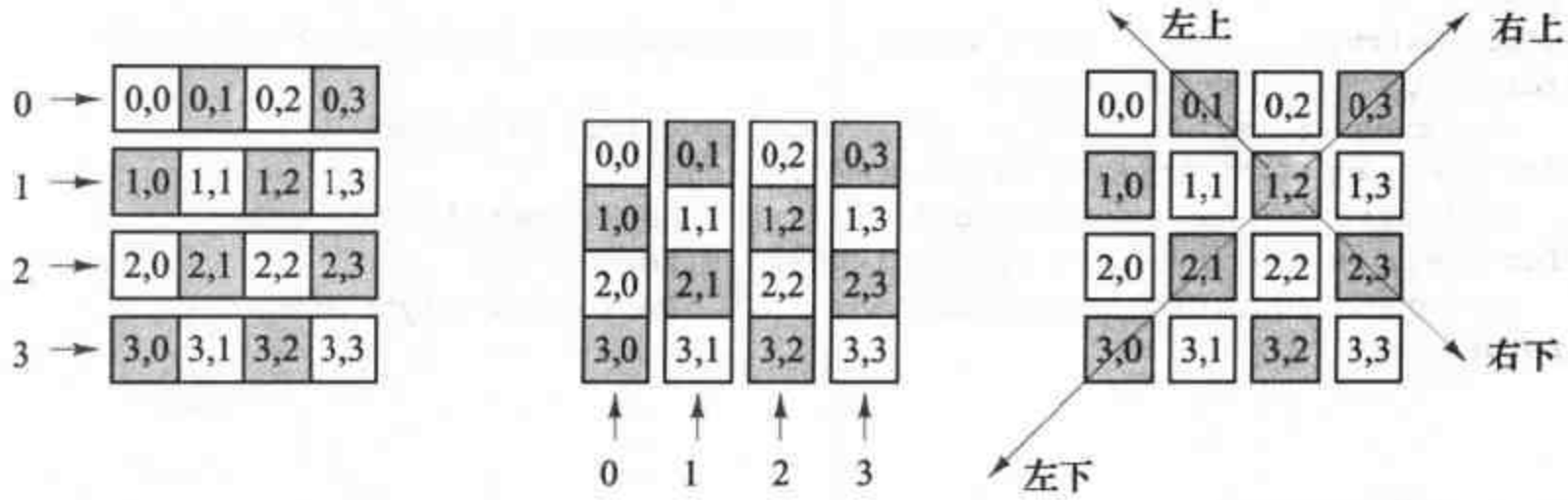
为了初始化 Queens 配置,需要有构造函数,此函数利用它的参数来设置棋盘的大小:

```
Queens::Queens(int size)
/* Post: The Queens object is set up as an empty configuration on a chessboard with size
         squares in each row and column. */
{
  board_size=size;
  count=0;
  for (int row=0; row<board_size; row++)
    for (int col=0; col<board_size; col++)
      queen_square[row][col]=false;
}
```

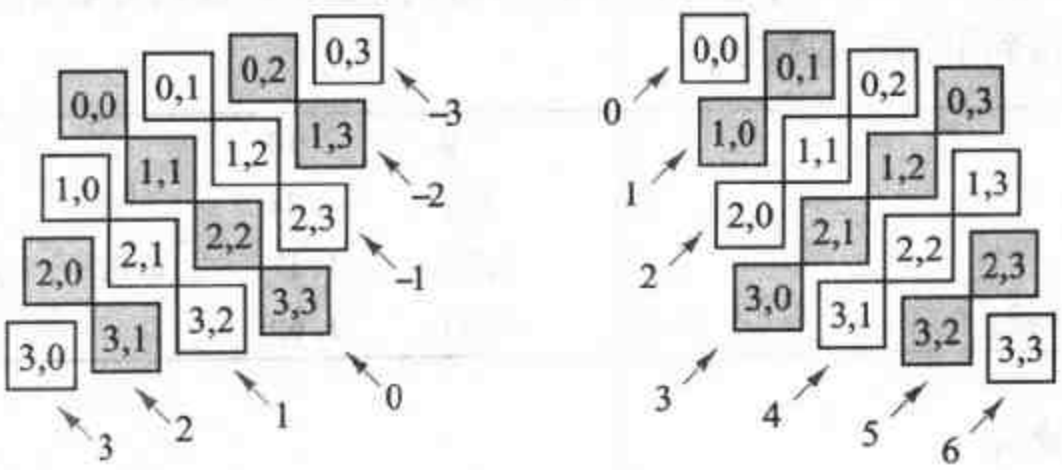
程序中已摆放的皇后数已设置为 0。每当这个构造函数执行时,都会定义一个 Queens 对象,并且同时指定了一个整型参数,就如在主程序中一样。

最后,必须编写一个方法来检查棋盘的首个空行中某特定列位置上的格子是否被一个

(或多个)已在配置中的皇后所设防。为完成这一任务,必须搜索此格子所在的列和两条对角线。对列的搜索很简单,但查找对角线要求更细致的下标计算。见图 5.14 中 4×4 棋盘的情况。



(a) 行　　(b) 列　　(c) 对角线方向



差 = 行 - 列　　和 = 行 + 列

(d) 向下的对角线　　(e) 向上的对角线

图 5.14　分成部分的棋盘

我们能将棋盘的某个格子所形成的对角线看成相当于 4 个对角线方向,可以分别称它们左下对角线(它从原格子向下并向左)、右下对角线、左上对角线和右上对角线。

首先考虑左上对角线,如图 5.14(c)部分所示。如果从棋盘上位置[row][col]的格子开始,则属于左上对角线的格子具有位置格式[row-i][col-i],其中 i 为正整数。左上对角线必须在棋盘的上部边缘或棋盘的左侧边缘结束,其中上部边缘为 row-i ==0,左侧边缘为 col-i==0。因此通过一个 for 循环,让 i 从 1 开始递增,直到条件 row-i >=0 和 col-i>=0 中的任一个不成立,可以列出左上对角线的格子。

类似的循环描绘了棋盘上给定格子所形成的其他三个对角线。但是,当检查棋盘的首个空行中某特定格子是否被设防时,决不必检查下部的两条对角线,因为下部的格子自动是空着的。因此,在方法 unguarded 的代码中仅反映出上部对角线的情形,随后即是此方法的代码。

```
bool Queens::unguarded(int col) const
/* Post: Returns true or false according as the square in the first unoccupied row (row
        count) and column col is not guarded by any queen. */
{
  int i;
  bool ok=true;        //  turns false if we find a queen in column or diagonal
  for (i=0; ok && i<count; i++)
    ok=!queen_square[i][col];   //  Check upper part of column
  for (i=1; ok && count-i>=0 && col-i>=0; i++)
    ok=!queen_square[count-i][col-i];   //  Check upper-left diagonal
  for (i=1; ok && count-i>=0 && col+i<board_size; i++)
    ok=!queen_square[count-i][col+i];   //  Check upper-right diagonal
  return ok;
}
```

5.3.6 复查和细化

刚才已完成的程序对“八皇后”问题已足够了，结果表明在 8×8 棋盘上摆放 8 皇后共有 92 种解答。然而，当对较大的棋盘运行这个程序时，会发现它很快开始消耗大量的时间。例如一次运行可以得到下面这些数字：

大小	8	9	10	11	12	13
解答数	92	352	724	2680	14200	73712
时间(秒)	0.05	0.21	1.17	6.62	39.11	243.05
每个解答所需时间(毫秒)	0.54	0.60	1.62	2.47	2.75	3.30

* * * 快速增长的时间

从上表可看出，解答数随着棋盘的大小快速增长，而时间甚至增长得更快，因为每个解答所需的时间也随着棋盘大小增长。如果想得到更大型棋盘的结果，必须找到一种更有效的程序，否则将花费大量的计算机时间。

现在来看我们的程序在何处花费的时间最多。进行递归调用和回溯占用了大量时间，但这些时间反映了求解此问题的基本方法和大量解的存在。在方法 unguarded()中的几个循环也会需要相当多的时间。现在就来看看能否消除这些循环，也就是说，是否可能不用搜索格子的行、列和对角线而确定这个格子是否被设防。

* * * 第一遍细化

实现它的一种办法是修改表示棋盘的正方形数组中存储的数据。可以使用这个数组记录皇后所设防的所有格子，而不是仅记录那些被皇后占用的格子，则检查一个格子是否未设防就简单了。由于一个格子可能被多于一个皇后设防，因此稍做修改会有助于回溯。对每个格子，可以保存设防它的皇后的计数。那么当插入一个皇后时，将同一行、列和对角线上的所有格子的计数加 1。当删除一个皇后时，只要将这些计数减 1。

此方法的程序设计留作一个项目。注意此方法虽然比前一个更快，但它仍然需要循环来更新每个格子的设防计数。实际上，再多一些思考，可以消除所有这些循环。

* * * 第二遍细化

关键思想是注意到棋盘上的每行、每列和每条对角线能包含至多一个皇后。(鸽巢原理

表明:在解答中,所有的行和所有的列都被占据,但并非所有的对角线会被占据,因为对角线比行和列更多。)

因此通过使用3个bool数组来记录空着的格子:col_free,upward_free和downward_free,这时将左下到右上的对角线看成向上的,并将左上到右下的对角线看成向下的。(见图5.14的(d)和(e)部分。)由于皇后在棋盘上的摆放是从第0行开始,每次一行,因此不必使用一个显式数组来找出哪些行是空的。

最后,为了打印配置,需要知道每一行中皇后的列号,对它可以使用一个用列作为下标的整型值数组。

*** 不带循环的程序

注意现在甚至可以不用表示棋盘的正方形数组来求解整个问题,并且除了初始化几个"free"数组的循环外,不用其他任何循环。因此修订后的程序所需要的运行时间近似地反映了回溯中探查的步骤数。

如何标识沿着单条对角线的格子?沿着最长的向上的对角线,各元素的下标为:

[board_size-1][0],[board_size-2][1],… ,[0][board_size-1]

这些下标具有一个特性,即行下标和列下标的和总是board_size-1。它说明(如图5.14(e)部分所示)沿着任何向上的对角线,行和列的下标将有一个常量和。这个和的范围是从0到2×board_size-2,其中0对应左上角长度为1的向上的对角线,2×board_size-2对应右下角长度为1的向上的对角线。因此可以给向上的对角线从0到2×board_size-2编号,则在向上的对角线中第i行第j列的格子编号为$i+j$。

类似地,沿着向下的对角线(如图5.14的(d)部分所示),行与列下标的差是一个常量,范围是从-board_size+1到board_size-1。因此我们能用0到2 × board_size-1来给向下的对角线编号,则在向下的对角线中,第i行第j列中的格子的编号为$i-j$+board_size-1。

在做出所有这些决定后,现在可以正式地说明修订后的Qucens数据结构。

```
class Queens {
public:
  Queens(int size);
  bool is_solved() const;
  void print() const;
  bool unguarded(int col) const;
  void insert(int col);
  void remove(int col);
  int board_size;
private:
  int count;
  bool col_free[max_board];
  bool upward_free[2*max_board-1];
  bool downward_free[2*max_board-1];
  int queen_in_row[max_board];   //   column number of queen in each row
};
```

下面提供修订后的Queens类的方法,并以此完成全部程序。首先是构造函数:

```
Queens::Queens(int size)
/* Post: The Queens object is set up as an empty configuration on a chessboard with size
        squares in each row and column. */
```

```
{
  board_size=size;
  count=0;
  for (int i=0; i<board_size; i ++)col_free[i]=true;
  for (int j=0; j<(2*board_size-1); j++)upward_free[j]=true;
  for (int k=0; k<(2*board_size-1); k++)downward_free[k]=true;
}
```

它类似于第 1 版的构造函数，除了现在是将所有的列和对角线都标记为未设防的，而不是初始化格子数组。

方法 insert()对对角线编号的约定进行编码。

```
void Queens::insert(int col)
/* Pre:  The square in the first unoccupied row (row count) and column col is not guarded
         by any queen.
   Post: A queen has been inserted into the square at row count and column col;count has
         been incremented by 1.*/
{
  queen_in_row[count]=col;
  col_free[col]=false;
  upward_free[count+col]=false;
  downward_free[count-col+board_size-1]=false;
  count++;
}
```

最后，方法 unguarded()仅需要测试包含某特定格子的列及两条对角线是否未被设防。

```
bool Queens::unguarded(int col) const
/* Post: Returns true or false according as the square in the first unoccupied row (row
         count) and column col is not guarded by any queen.*/
{
  return col_free[col]
         && upward_free[count+col]
         && downward_free[count-col+board_size-1];
}
```

注意 unguarded()函数比起第 1 版来不知要简单多少，实际上可以注意到，除了构造函数中的初始化代码以外，在任何方法中都没有循环。

剩余的方法 is_solved()，remove()和 print()确实可留作练习。

下面的表给出有关 n-皇后问题的新程序的性能信息。为方便比较，使用与前面程序的相同的测试条件，在相同的机器上产生这些数据。时间数据表明对 8-皇后问题，新的程序比老的程序运行大约快 5 倍。当棋盘大小增长时，我们希望新的程序比老的程序快得更多。的确，对 13-皇后问题，新程序大约快 7 倍。

大小	8	9	10	11	12	13
解答数	92	352	724	2680	14200	73712
时间(秒)	0.01	0.05	0.22	1.06	5.94	34.44
每个解答所需时间(毫秒)	0.11	0.14	0.30	0.39	0.42	0.47

5.3.7 关于回溯的分析

在本节结束时，我们来估计程序所做的工作量。

1. 回溯的有效性

首先来看看回溯与列举所有可能性进行比较能够节省多少工作。为得到用数字表示的结果，仅考虑8×8的案例。如果采用一种天真的方法，编写程序首先将所有8个皇后摆放在棋盘上，并随后滤去不合理的配置，则所需要探查的配置数相当于从64个位置中选择8个，即：

$$\binom{64}{8} = 4\ 426\ 165\ 368.$$

每行仅能有一个皇后的事实马上将这个数削减到$8^8 = 16\ 777\ 216$。

* * * 削减后的数目

这个数仍很大，但我们的程序不会探查这么多格子。实际上，它滤去那些列或对角线已被设防的格子。在每一列仅有一个皇后的要求将这个数再削减为$8! = 40\ 320$。这个数字对计算机来说是完全可以管理的，而程序考虑的情况的实际数目还会比这个数小得多，因为在早期行中被设防的对角线上的格子会被马上滤去，而不必再做无用的尝试来填入后面的行。

* * * 回溯的有效性

这种行为概括了回溯的有效性：被发现不再可能的位置避免了后续无用路径的探查。

表达回溯这种行为的另一种方式是考虑递归函数solve_from的递归调用树，图5.15中给出了其中一部分。树中所示的两种解答与图5.12中所示的解答是一样的。形式上看起来，树中的每个结点可能最多有8个孩子，它们分别对应于用new_col的8个可能值做出的对solve_from的递归调用。但即使在靠近根的层中，其中的大部分分支是不可能的，而在上层移去一个结点就移去了它的大量后代。回溯是对递归树进行剪枝，使其达到可管理规模的最有效的方法。

2. 下界

另一方面，对n-皇后问题，回溯所做的工作量仍然随n快速增长，让我们对究竟有多快得到一个十分粗略的概念。当在棋盘中某一行摆放一个皇后时，注意到它在棋盘上随后的每一行中将排除至少3个位置(它所在的列、上对角线和下对角线)。对第一行，回溯方法会在n个位置上探查摆放皇后的可能性。对第二行，它必须探查至少$n-3$个位置，对第三行至少$n-6$，以此类推。因此，为了在前$n/4$行中的每一行摆放一个皇后，回溯探查的位置数最少为$n(n-3)(n-6)\cdots(n-3n/4)$。这些因子中的最后一个是$n/4$，其他的都较大，并且有$n/4$个因子。因此，仅填充最初的$n/4$行，回溯必须探查超过$(n/4)^{n/4}$个位置。

* * * 指数级增长

为得到这个数随n增长多快的概念，回忆一下汉诺塔，对n个盘子需要2^n步，并注意当n增加时，$(n/4)^{n/4}$比2^n增长更快。为证明这一点，仅需要了解$\log((n/4)^{n/4})/\log(2^n) = \log(n/4)/4\log(2)$。而当$n$增长时，明显地这个比率无限增长。我们说$2^n$按指数增长，而

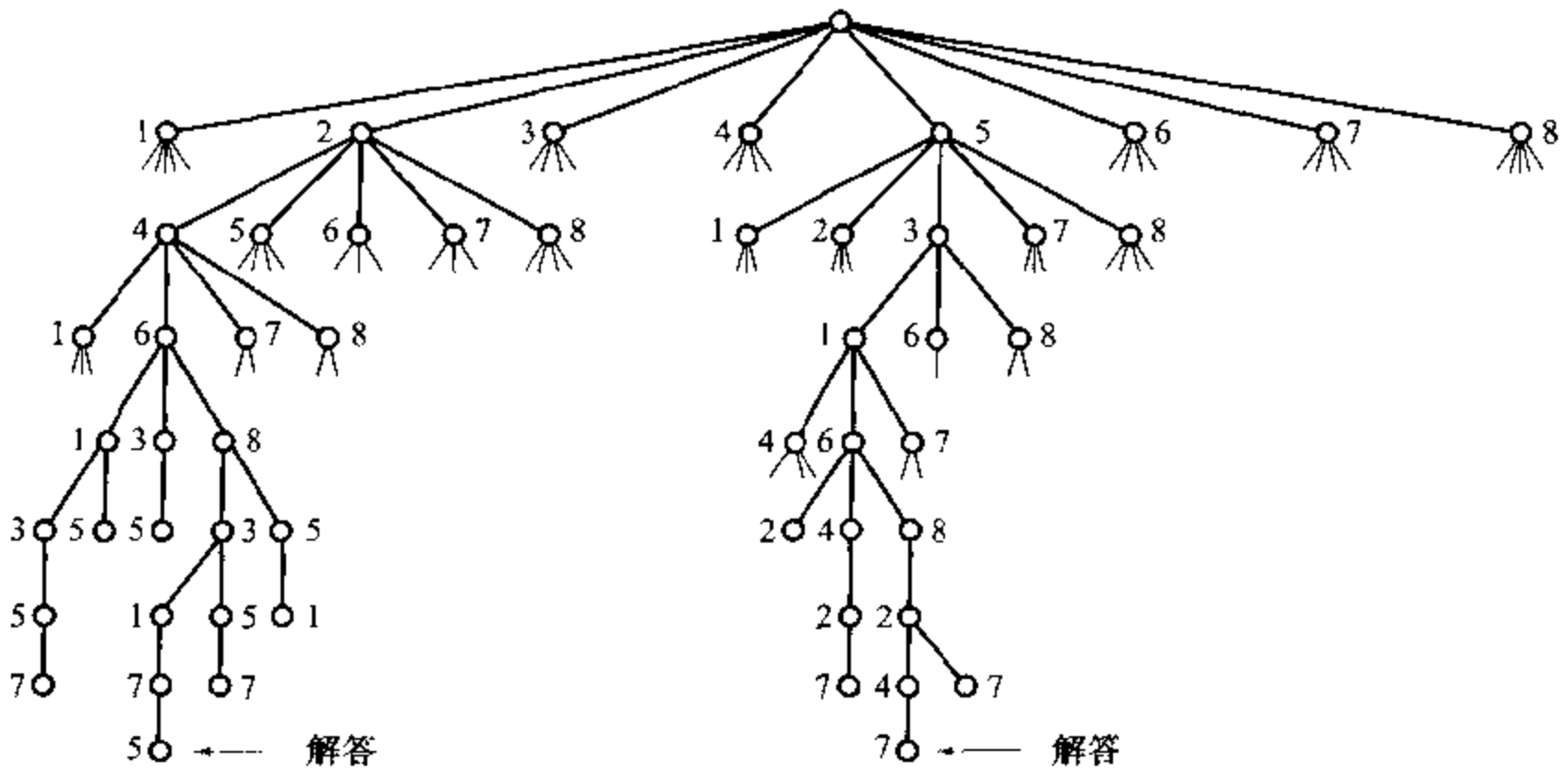


图 5.15 八皇后问题的部分递归树

$(n/4)^{n/4}$增长得更快。因此当 n 增长时，n-皇后问题的回溯会慢得无法想像。

3. 解的数目

* * * 未解决的问题

注意我们尚未证明对较大的 n，不可能用计算机打印出 n-皇后问题的所有解，而只是证明回溯做不了。或许可能有某种其他巧妙的算法能比回溯更快地显示出这些解答，但事实并非如此。现在能(见参考书目)证明不能将 n-皇后问题的解答数限定在 n 的任何多项式中。实际上看来，甚至不能将它限制在指数形式 k^n 的任何表达式中(此处 k 为一个常量)，但对它的证明还是一个未解决的问题。

练习 5.3

E1. 函数 solve_from 中递归的最大深度是多少?

E2. 用下面棋盘上 5 个皇后的部分配置开始，构造函数 solve_from 在加入剩余 3 个皇后时将考虑的所有情况的递归树。在函数要回溯的地方停止并移去原来 5 个皇后中的一个。

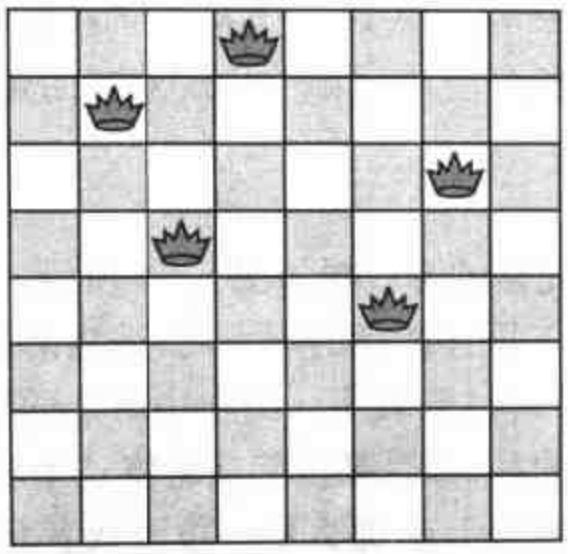
E3. 通过手工执行回溯，找到在 5×5 棋盘上摆放 5 个皇后问题的所有解答。可以在第一行使用左右对称的方法，在第一行仅考虑在第 1，第 2 或者第 3 列摆放皇后的可能性。

程序设计项目 5.3

P1. 在计算机上运行八皇后程序：

(1)编写尚缺的 Queens 方法。

(2)通过引入一个计数器，在 solve_from 每次启动的时候递增此计数器，准确地找出要



探查的棋盘位置数。(注意在启动检查设防的格子前摆放所有 8 个皇后的方法等价于对 solve_from 的 8 次调用。)

(3)用 4～15 范围内的皇后数运行程序,尝试找出一个数学函数,它作为皇后数的函数,近似于探查的位置数。

P2. 超皇后不仅能走皇后的棋步,也能走马的棋步(见项目 P4。)修改项目 P1,在其中使用超皇后而不是普通皇后。

P3. 用数组描述矩形迷宫,在数组中指示迷宫的路径和墙。编写一个递归程序找出一条通过迷宫的道路。

P4. 另一个棋盘难题(这一难题被认为由高斯在四岁时求解)是找出马的走动序列,它对棋盘上每个格子仅访问一次。马的走动是在水平方向或者垂直方向上跳两个位置并在与其成直角的方向上再走一个位置(译者注:中国象棋有一句行话称为"马跳日")。这种走步可以用以下方式完成:将 x 设置为 1 或 2,将 y 设置为 $3-x$,然后通过 $\pm x$ 修改第一个坐标,通过 $\pm y$ 修改第二个坐标(假定结果得到的位置仍在棋盘上)。编写一个回溯程序,输入初始位置并搜索马从给定位置开始,走遍每个格子一次且仅走一次时的周游访问。如果发现程序运行太慢的话,一种比较好的办法是排出从给定位置出发能到达的格子的列表,使得程序首先走到具有最小可达性的格子,即从该格子出发到还未被访问过的格子有最少的马步走法。

P5. 修改 P4 中的程序,使它按马所访问的顺序对棋盘格子编号,马出发的位置编号为 1。修改此程序使得它能够找到一个魔幻的马的周游访问,即此周游访问所得到的格子编号形成一个幻方。(见 1.6 节项目 P1(a)对幻方的定义。)

5.4 树结构的程序:在游戏中预测

在脑力技能游戏中,能够提前预测几步内将发生什么事情的人胜过那些只能看到眼前利益的竞争对手。本节设计一个计算机算法,它在玩游戏时能提前查看几步走法,这个算法能用树来描述。之后本节将说明如何用递归对这种结构进行程序设计。

5.4.1 游戏树

可以用树来描绘可能的走步序列,树的根表示初始情况,来自根的分支表示第一位游戏

者能下出的有效走法。在接下来的一层中，分支对应于各种情况下第二位游戏者的有效走法，以此类推，来自偶数层顶点的分支表示第一位游戏者的走法，来自奇数层顶点的分支表示第二位游戏者的走法。

* * *8

简单的 Eight 游戏对应的完整的游戏树如图 5.16 所示。在这个游戏中，第一位游戏者选择数字 1,2,3 当中的一个。在每种选择的下一轮，适当的游戏者选择 1,2,3 当中的一个数，但他不能选择前一游戏者已选过的数字。图中用所选的数字给树的分支打上标签。在游戏中维持这些数字连续相加的和，如果某位游戏者将这个和恰好凑成 8，那么这个游戏者获胜。如果某位游戏者使这个和大于 8，那么另一位游戏者获胜。Eight 游戏不可能出现平局。在图中，F 代表第一位游戏者获胜，S 代表第二位游戏者获胜。

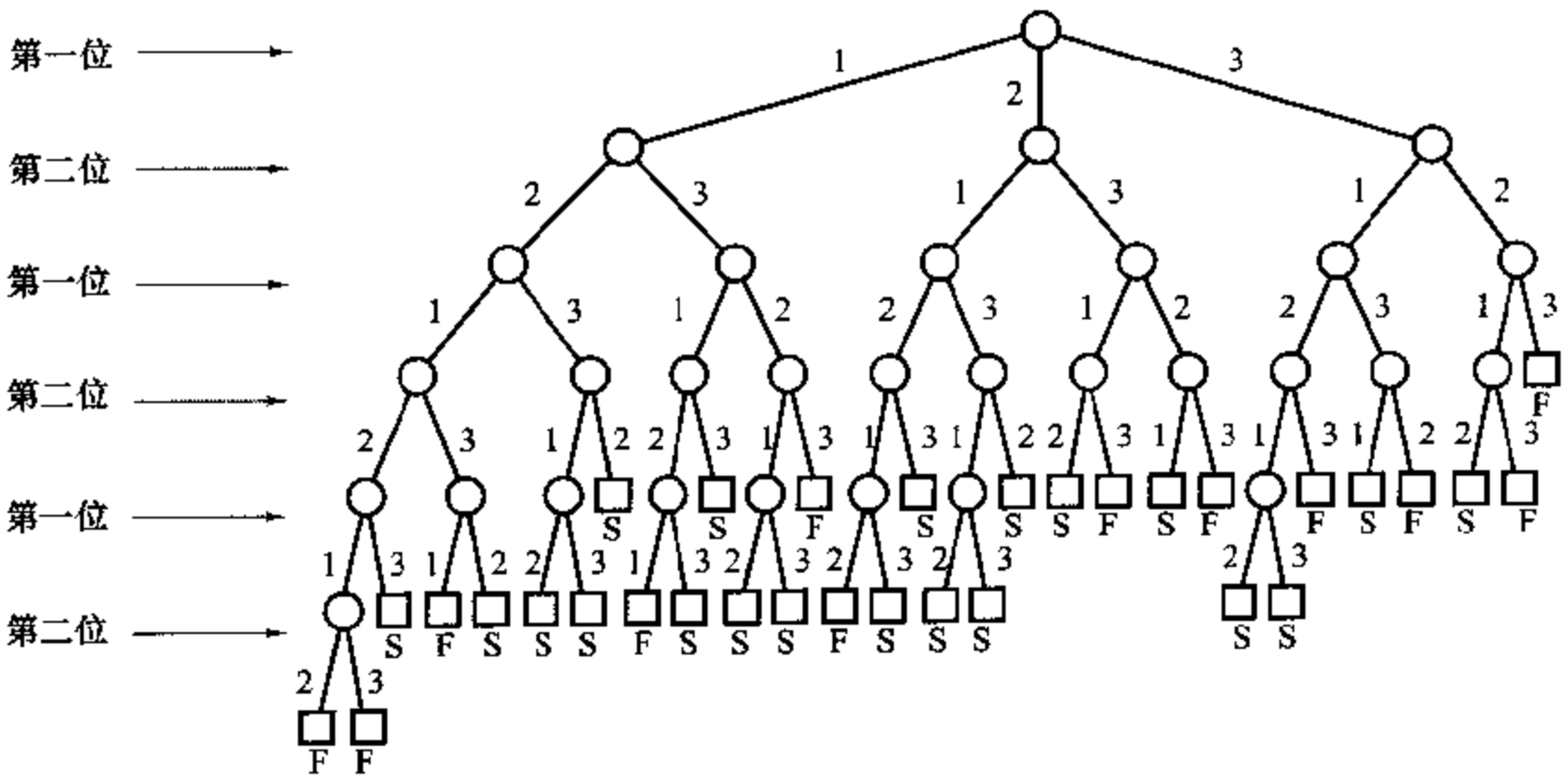


图 5.16 Eight 的游戏树

即便是 Eight 这样的小游戏也产生相当大的树，像国际象棋或围棋这些有实际趣味的游戏，所对应的游戏树将非常巨大，不可能探查所有的分支，在合理时间内运行的程序仅能检查树中当前顶点下的几层。玩这种游戏的人也不可能看到至游戏结束时的每种可能，但他们能做出明智的选择，即使他们并不能保证能赢，但以他们的经验，可以认识到游戏中的某些情形比其他情形更好。

* * *评价函数

因此，对我们推荐用计算机来玩的任何游戏，需要某种评价函数来检查当前的局面并返回一个估计其利益的整数。为明确，我们假定较大的数字反映对第一位游戏者有利的情形，而较小的(或较负的)数字表明对第二位游戏者有利。

5.4.2 极小极大方法

图 5.17 中出现的是一个虚构游戏的部分树。由于是在预测，我们仅在树的树叶处(即，在游戏中不用再更进一步查看的那些位置)需要评价函数，然后希望利用这个信息选择一种

走法。我们将游戏树中的树叶画作方格，并将其他结点画作圆圈。因此图 5.16 仅给出了以方格画出的结点的值。

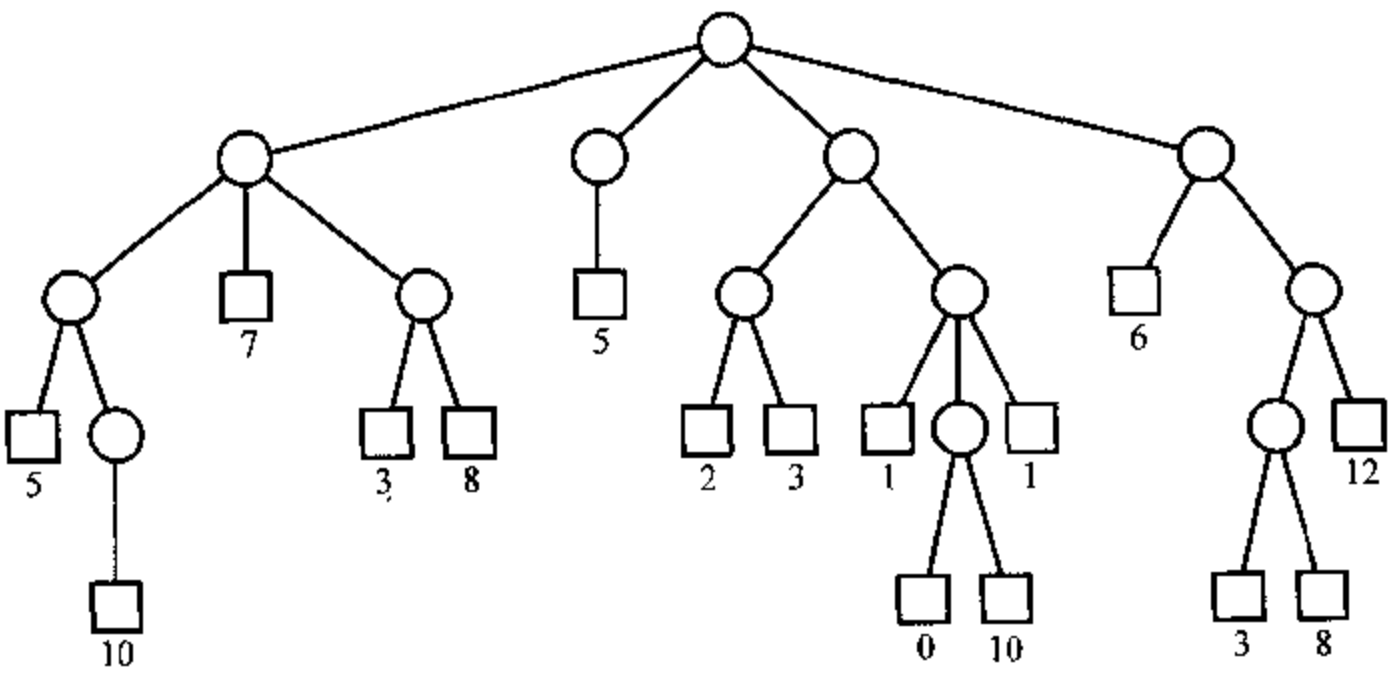


图 5.17　树叶赋值的游戏树

最终选择的走法是直接来自树的最顶层的根的一个分支。我们以走出第一步棋的游戏者的观点来看评价函数，这意味着这个游戏者选择尽可能的极大值。在接下来向下的一层，另一个游戏者会选择尽可能的极小值，以此类推。

＊＊＊上溯树

通过从树的底部向上计算，可以对所有的顶点赋值。现在从图 5.17 中树的左下边开始，上溯此图中部分路线的这个过程。第 1 个未标注的结点是在标注 10 的方格上面的圆圈。由于在此圆圈结点处没有走法选择，因此它的值必须也是 10。它的双亲结点有 2 个孩子，现在分别标注了 5 和 10。这个双亲结点在树的第 3 层上。即，这代表了第一位游戏者的走法，他希望这个值极大化。因此，这位游戏者会选择值为 10 的走法，故这个双亲结点的值也是 10。

接下来在树中往上移一层到有 3 个孩子的结点。现在已知最左侧的孩子的值为 10，第 2 个孩子的值为 7。最右侧的孩子的值应是其两个孩子的值 3 和 8 中最大的，因此是 8。这 3 个孩子的双亲结点是在第 2 层，即这代表了希望这个值极小化的游戏者的走法。因此这位游戏者会选择 3 种可能性中中间的走法，此结点的值因此是 7。

如此继续这个过程。我们花上一些时间就可以完成图 5.17 中所有结点的评价，评价结果在图 5.18 中给出。目前情形对应的结果值是 7，当前(第一位)游戏者会选择最左侧的分支作为最佳走法。

＊＊＊极小极大方法

浅灰色显示的虚线将在后面的一个项目中加以解释。结果表明，通过记录目前发现的极大值和极小值，不必去评价游戏树中的每个结点，并且在图 5.18 中，不必去评价围在虚线中的结点。由于在游戏树的评价中，我们交替使用极小值和极大值，因此这个过程称为极小极大方法。

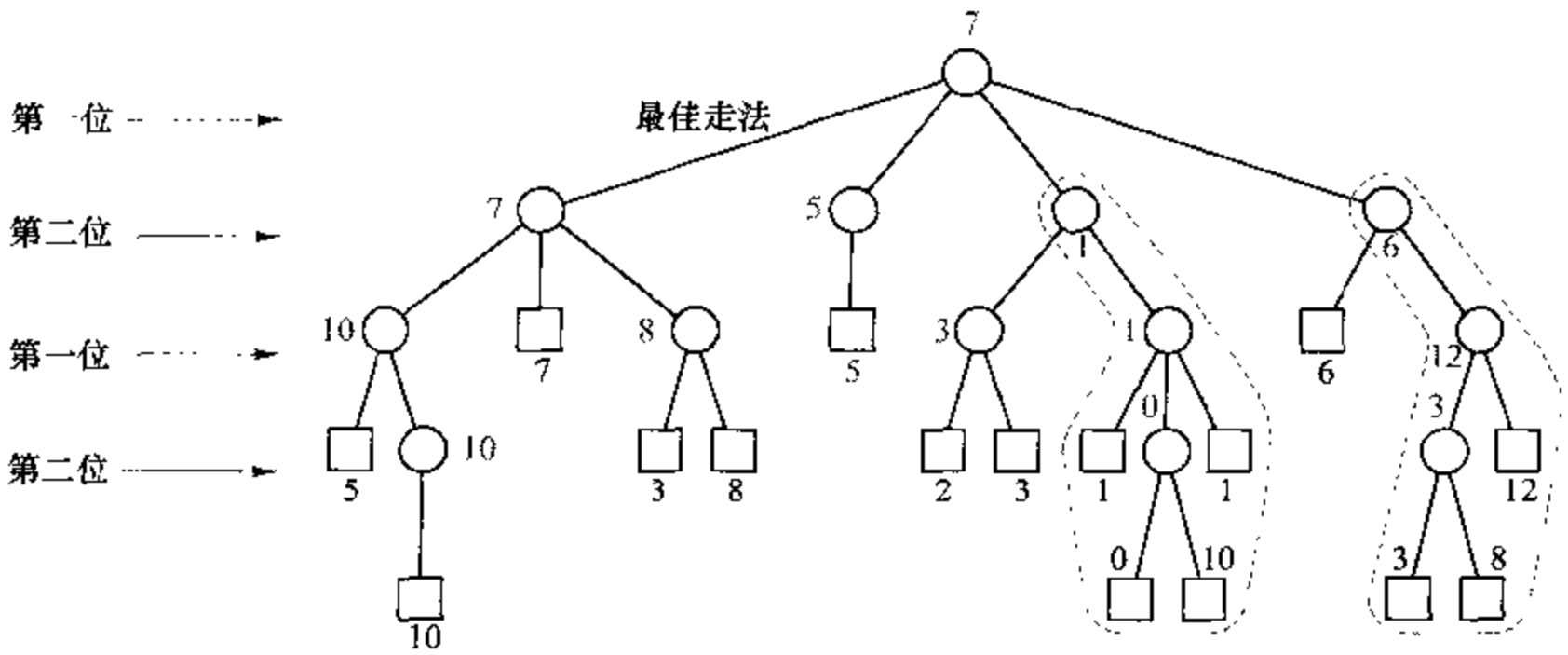


图 5.18 游戏树的极小极大评价

5.4.3 算法设计

接下来让我们看看如何能用形式化的算法表达极小极大方法，并在游戏程序中进行预测。我们希望编写一个通用的算法，可以将它用于任何两个游戏者的游戏。

我们的程序需要访问关于特定游戏的信息，假设这些信息被收集在称为 Move 和 Board 的类的实现中。类型 Move 的对象代表单步的游戏走法，类型 Board 的对象代表单个的游戏位置。其后，我们将对并字游戏（连三字棋）实现这些类的描述。

对于 Move 类，我们仅需要构造函数。其中有一个构造函数用来创建由客户程序指定的 Move 对象，并且第二个构造函数是默认的构造函数，用来创建空的 Move 对象。我们也假设 Move 对象（及 Board 对象）能作为值参数传递给函数，并且能用赋值运算符（即运算符=）安全地复制它们。

对于 Board 类，我们明确要求以下方法：初始化 Board；检测游戏是否结束；用参数传递的走法下一步棋；评价位置；以及提供当前全部有效走法的列表。

*** 有效的走法

方法 legal_moves 给出当前的走法选项，它需要一串参数来传递其结果。我们选择了几种列表数据结构来存放这些走法。列表的顺序将在更后阶段的预测中被探查，但这个顺序并不重要，因此能将它们保存在任何形式的列表中。为了程序设计的简单性，我们使用栈结构，栈中的元素是走法。因此，为了利用早期的 Stack 实现，我们要求下面的定义：

```
typedef Move Stack_entry;
```

*** 比较走法的值

我们还需要其他两个方法，在为下棋者选择最有利的走法时会用到它们，下棋者定义为必须走出下一步棋的游戏者。其中的一个方法称为 better，它使用两个整型参数，并且当下棋者更喜欢第一个参数而不是第二个参数给出的游戏值时，better 函数返回非零的结果。

*** 找出最坏情况下的值

另一个方法是 worst_case，它返回下棋者明确喜欢的一个预定的常数值，这个值比任何

可能的游戏位置的值更小。虽然我们不用与用户沟通就能分析游戏，运用我们的分析来玩游戏的任何程序需要 Board 方法打印出存储的位置并打印出游戏指令。

* * * 保持 Board 不改变

就像国际象棋游戏者除了下棋外不去碰棋盘上的棋子，我们要求 Board 方法(除了下棋以外)保持 Board 数据成员不改变。国际象棋中碰子走子的规则有助于向裁判或观棋者保证游戏公平进行。而类似地，我们对类 Board 所做的保护可以使利用这个类的程序设计员放心。我们已经知道，在 C++中将修改符 const 附在一个方法或成员函数的参数列表后面，保证函数不会修改相应对象中的数据成员。因此我们对 Board 类的定义将具有如下形式：

```
class Board {
public:
  Board();                  //  constructor for initialization
  int done() const;         //  Test whether the game is over.
  void play(Move try_it);
  int evaluate() const;
  int legal_moves(Stack &moves) const;
  int worst_case() const;
  int better(int value, int old_value) const;
                            //  Which parameter does the mover prefer?
  void print() const;
  void instructions() const;
  /* Additional methods, functions, and data will depend on the game under considera-
     tion.*/
};
```

注意，Board 类的数据成员需要同时记录棋盘位置和哪位游戏者是下棋者。

* * * 终止

在编写用预测评价游戏树的函数之前，我们需要决定预测算法何时停止进一步查看。对于具有合理复杂性的游戏，我们必须建立许多层的深度 depth，超过它就不再搜索。另一个终止条件是游戏结束：检测到 Board::done()返回 true 值。在树中预测的基本任务现在能用下面的递归算法来描述。

* * * 算法概要

look_ahead at game (Board 对象)；
if 递归终止(也就是 depth==0 或 game.done())
返回此位置的评价
else
对每个有效的 Move
通过走出 Move 创建一个新的 Board，并递归地预测与另一游戏者的最佳后续 Move 对应的游戏值；
从循环中发现的值当中为下棋者选择最佳的选项；
返回对应的 Move 和游戏值作为结果；

5.4.4 细化

5.4.3 节的函数概要产生下面的递归函数：

```
int look_ahead(const Board &game, int depth, Move &recommended)
/* Pre: Board game represents a legal game position.
   Post: An evaluation of the game, based on looking ahead depth moves, is returned. The
         best move that can be found for the mover is recorded as Move recommended.
   Uses: The classes Stack, Board, and Move, together with function look_ahead recursi-
         vely. */
{
   if (game.done() || depth==0)
      return game.evaluate();
   else {
      Stack moves;
      game.legal_moves(moves);
      int value, best_value=game.worst_case();
      while(!moves.empty()){
       Move try_it, reply;
       moves.top(try_it);
       Board new_game=game;
       new_game.play(try_it);
       value=look_ahead(new_game, depth-1, reply);
       if (game.better(value, best_value)){
                        //  try_it is the best move yet found
         best_value=value;
         recommended=try_it;
       }
       moves.pop();
      }
      return best_value;
   }
}
```

引用参数 Move recommended 用于返回一种推荐的走法(除非游戏结束或者搜索深度为 0)。因为我们不希望在函数中修改 Board,所以引用参数 Board game 可以说明为值参数,但是为了避免可能高代价的复制操作,我们将 game 作为常量引用参数传递。注意,因为应用对象 Board game 的惟一方法 Board 已用修改符 const 声明,因此编译器能保证它不被函数 look_ahead 修改。如果我们在类 Board 的定义中没有早做准备,那么将参数 Board game 作为常量传递将是非法的。

5.4.5 井字游戏

在本节结束之时,我们将在井字游戏(连三字棋)中给出 Board 和 Move 类的实现,在这里,这些类只是由游戏规则的形式化实现组成。

* * * 主程序

我们把利用这些类的主程序的编写留作一个项目,它们采用函数 look_ahead 来玩井字游戏。在这样的程序中,可以遵循许多选项:计算机可作为人类的对手,给出位置的完整分析,或者给出它对两个人类游戏者的走法的评价。

我们将井字游戏的网格表示为一个 3×3 的整型数组,并用 0 表示空的格子,用 1 和 2 分别表示第一位和第二位游戏者占据的格子。

在 Move 对象中,我们仅存储网格上的格子的坐标。对有效的走法,这些坐标将在 0～2

之间。我们不打算封装 Move 对象，因为它们只不过是数据值集合的拥有者，我们因此得到 Move 类的如下实现：

```
//  class for a tic-tac-toe move
class Move {
public:
  Move();
  Move(int r, int c);
  int row;
  int col;
};

Move::Move()
/* Post: The Move is initialized to an illegal, default value.*/
{
   row=3;
   col=3;
}
Move::Move(int r, int c)
/* Post: The Move is initialized to the given coordinates.*/
{
   row=r;
   col=c;
}
```

我们已看到，Board 类需要构造函数(用来初始化游戏)、print 方法和 instructions 方法(它们为用户打印出信息)、done 方法、play 方法和 legal_moves 方法(它们实现游戏的规则)、以及 evaluate 方法、better 方法和 worst_case 方法(它们判断各种走法的值)。我们发现另有一个辅助函数 the_winner 是有用的，它返回的值指示是否赢得胜利，如果是的话，则指示是哪位游戏者。

Board 类也必须存储数据成员来记录 3×3 数组中当前的游戏状态，并记录已走出了多少步。因此我们得出下面的类定义：

```
class Board {
public:
  Board();
  bool done() const;
  void print() const;
  void instructions() const;
  bool better(int value, int old_value) const;
  void play(Move try_it);
  int worst_case() const;
  int evaluate() const;
  int legal_moves(Stack &moves) const;
private:
  int squares[3][3];
  int moves_done;
  int the_winner() const;
};
```

*** 构造函数

构造函数只是用值 0 填充数组 squares，指示两位游戏者都还没有下出任何棋子。

```
Board::Board()
/* Post: The Board is initialized as empty. */
{
  for (int i=0; i<3; i++)
    for (int j=0; j<3; j++)
      squares[i][j]=0;
  moves_done=0;
}
```

* * * 下一步棋

我们将打印信息的方法留给用户作为习题，而接下来集中研究应用游戏规则的方法。为了下一步棋，我们仅需要重置 squares 中的一个值并更新计数器 moves_done 来记录已下了另一步棋。计数器 moves_done 的值用于计算这一步棋究竟应归于游戏者 1 和游戏者 2 中的哪一位。

```
void Board::play(Move try_it)
/* Post: The Move try_it is played on the Board. */
{
  squares[try_it.row][try_it.col]=moves_done % 2+1;
  moves_done ++;
}
```

* * * 决定胜者

如果任一游戏者获得胜利，辅助函数 the_winner 将返回一个非零的结果。它依次测试 Board 的全部 8 条可能的线路。

```
int Board::the_winner() const
/* Post: Return either a value of 0 for a position where neither player has won, a value
        of 1 if the first player has won, or a value of 2 if the second player has won. */
{
  int i;
  for (i=0; i<3; i++)
    if (squares[i][0]&&squares[i][0]==squares[i][1]
                     && squares[i][0]==squares[i][2])
      return squares[i][0];
  for (i=0; i<3; i++)
    if (squares[0][i]&&squares[0][i]==squares[1][i]
                     &&squares[0][i]==squares[2][i])
      return squares[0][i];
  if (squares[0][0]&&squares[0][0]==squares[1][1]
                   &&squares[0][0]==squares[2][2])
      return squares[0][0];
  if (squares[0][2]&&squares[0][2]==squares[1][1]
                   &&squares[2][0]==squares[0][2])
      return squares[0][2];
  return 0;
}
```

或者当已下完 9 步棋，或者当一个或另一个游戏者获胜时，游戏结束。（只有当 9 个格子都被填满时，程序才承认游戏是一场平局。）

```
bool Board::done() const
/* Post: Return true if the game is over; either because a player has already won or
```

```
        because all nine squares have been filled. */
{
  return moves_done==9||the_winner()>0;
}
```

游戏者可用的有效走法仅是 squares 中值为 0 的格子。

```
int Board::legal_moves(Stack &moves)const
/* Post: The parameter Stack moves is set up to contain all possible legal moves on the
         Board. */
{
  int count=0;
  while(!moves.empty())moves.pop();
  for(int i=0; i<3; i++)
    for(int j=0; j<3; j++)
      if (squares[i][j]==0){
        Move can_play(i, j);
        moves.push(can_play);
        count++;
      }
  return count;
}
```

* * * 估计位置

我们现在考虑判断 Board 位置的值或潜在的 Move 的值的方法。当没有游戏者获胜时，我们最初估计 Board 位置为 0。然而，如果一个或另一个游戏者获胜，则我们应按照规则估计此位置，在规则中认为快速获胜非常好，认为快速失败非常差。当然，这种估计仅在 look ahead 的尾部应用，并且只要我们向前预测得足够远，那么它的粗糙性质就不是一个缺点了。

```
int Board::evaluate()const
/* Post: Return either a value of 0 for a position where neither player has won, a
         positive value between 1 and 9 if the first player has won, or a negative value
         between-1 and-9 if the second player has won, */
{
  int winner=the_winner();
  if(winner==1)return 10-moves_done;
  else if(winner==2)return moves_done-10;
  else return 0;
}
```

方法 worst_case 只能返回 10 或者 -10，因为 evaluate 总是产生 -9～9 之间的值。因此比较方法 better 仅需要比较 -10～10 之间的一对整数。我们将这些方法留作习题。

现在已大略叙述了井字游戏程序的大部分。将预测深度置为 9 的程序会玩一场完美的游戏，因为它总是能预测到位置估计是正确的情形，预测深度小于 9 的程序会出错，因为它可能以错误估计为 0 的位置的集合结束其预测。

练习 5.4

E1. 如图 5.16 所示，在 Eight 游戏中，对第一位游戏者的胜利赋值 +1，对第二位游戏者的胜

利赋值为-1，并用极小极大方法评价它的游戏树。

E2. Nim游戏的一个变种是用一堆杆子开始，游戏者每轮可从中取走1，2或者3根杆子，游戏者必须至少取走1根杆子(但不能超过堆中剩余的)。必须取最后一根杆子的游戏者失败。用下面的杆子数画出完整的游戏树：

(1)5根杆子　　(2)6根杆子

为树的树叶赋上适当的值，并用极小极大方法评价其他的结点。

E3. 画出井字游戏(连三字棋)的游戏树的最上3层(指示前两次走动)，并计算将在第4层中出现的顶点数。你可以利用对称来减少树的大小：例如，在第一次走动时，只显示3种可能(中央的格子，1个角或1个边格)，而不是所有的9种可能。在靠近根部利用对称可以进一步减少游戏树的大小。

程序设计项目5.4

P1. 编写主程序以及Move和Board类的实现，让计算机与人类对手玩Eight游戏。

P2. 如果足够详细地研究图5.17中的树，你其实会注意到：在极小极大过程中，不必得到所有顶点的值，因为这棵树的某些部分完全不可能出现最佳的走法。

假设我们从树的左下端开始，当一旦有了所有孩子的值，我们就为双亲顶点填入值。当做完树的左侧的两个主要分支中的所有顶点后，我们得到值7和5，因此最大值至少是7。当我们来看第1层中的下一个顶点及其左孩子时，会发现这个孩子的值是3。在此阶段是取极小值，因此赋给第1层中的这个双亲顶点的值不可能超过3(实际上是1)。由于3小于7，第一位游戏者将选择最左端的分支，而我们能够排除其他分支。这样，那些不用被评估的顶点显示在图5.18中的浅灰色虚线内。

* * *α—β剪枝

用这种办法消除顶点的过程称为α--β剪枝(alpha-beta pruning)。通常采用希腊字母α(alpha)和β(beta)表示所发现的边界点。

修改函数look_ahead，使它采用α—β剪枝减少要探查的分支数。通过玩几个游戏比较这两个版本的性能。

启示和易犯的错误

1. 在算法的初始设计中应自由地使用递归。当求解的主要步骤是由将问题简化的一个或几个更小的问题组成时，递归方法特别合适。
2. 研究几个简单的例子以了解是否应该使用递归以及递归是如何工作的。
3. 尝试用公式表示更普遍适用的方法。询问："如何能将这个问题分成几个部分?"或者"如何完成中间的关键步骤?"
4. 询问此问题的剩余部分能否以同样的或类似的办法完成，如果必要的话，修改你的方法以使它充分通用。
5. 找出停止条件，它指示此问题本身或问题的某个适当部分被完成。
6. 非常仔细地保证你的算法总会终止，并正确地处理简单的情况。

7. 分析递归算法的关键工具是递归树。画出适合于问题的一或两个简单例子的递归树。
8. 应对递归树加以研究，以发现递归是否不必要地重复工作，或者递归树是否表达了任务的有效划分。
9. 递归函数能正确地实现使用栈的迭代函数所完成的同样任务。仔细考虑是递归还是带栈的迭代将产生更清晰的程序，并且更深入地了解问题。
10. 如果空间因素重要的话，可以消除尾部递归。
11. 递归总是能够转换成迭代，但这条常规经常造成的后果是程序结构十分晦涩难懂。只有当程序设计语言本身无法避免产生这种晦涩时，我们才能容忍它，并且要求它有良好的文档。
12. 研究你的程序，看看它是否适合递归算法的一个标准范例，如分而治之、回溯或树结构的算法。
13. 使递归的使用适合问题的结构。当问题的条件能被完全理解时，所要求的算法的结构将更易于领会。
14. 对极端的情况总是保持小心，确信你的算法在到达任务结尾时能够优雅地终止。
15. 进行尽可能彻底的错误检查，确信函数在其前置条件中规定了所要求的每个条件，并且即使满足了这一条件，也要尽可能方便地避免函数受到许多违反其前置条件的情况的影响。

复习题

5.1

1. 定义术语“分而治之”。

5.2

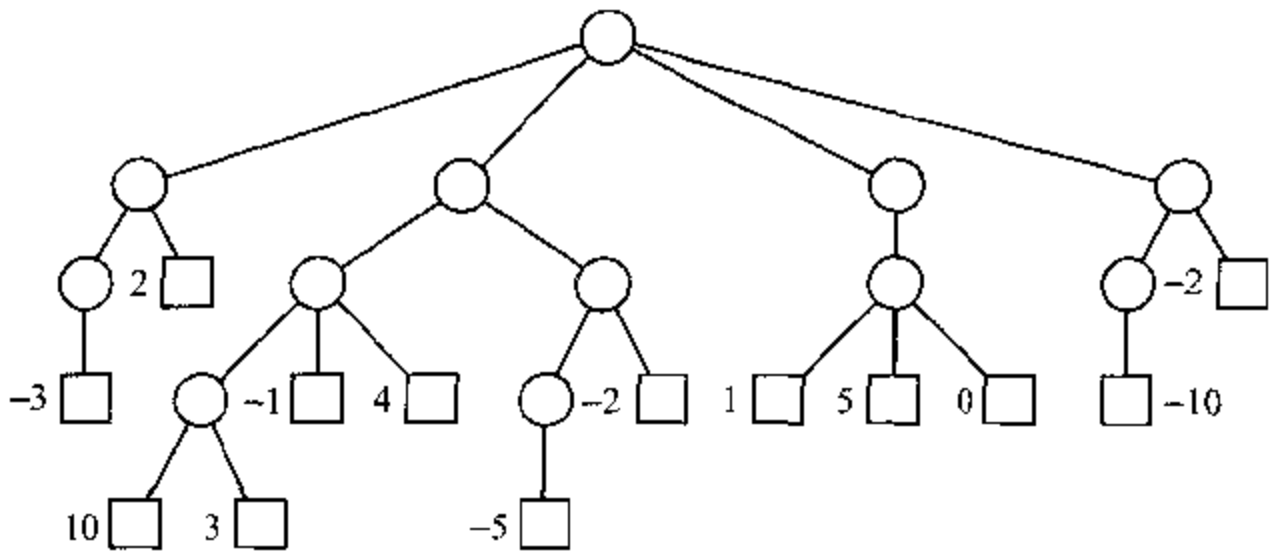
2. 列举两种不同的实现递归的方法。
3. 什么是重入程序?
4. 递归函数的时间需求与其递归树的关系如何?
5. 递归函数的空间需求与其递归树的关系如何?
6. 什么是尾部递归?
7. 描述递归树的形状和相应的递归程序的效率之间的关系。
8. 设计递归算法的主要阶段是什么?
9. 什么是并发?
10. 计算机系统在实现递归函数调用时需要保存哪些重要的信息?
11. 消除尾部递归对节省时间还是对节省空间更为重要?

5.3

12. 将回溯作为一种问题求解方法加以描述。
13. 陈述鸽巢原理。

5.4

14. 解释找出游戏值的极小极大方法。
15. 使用极小极大方法判断下面游戏树的值。



进阶参考书目

有两本书对递归做了全面的介绍，并带有许多例子，可以作为本书的极好的补充材料，它们是：Eric S. Roberts, *Thinking Recursively*, John Wiley & Sons, New York, 1986, 179 pages.

汉诺塔已是众所周知，并且出现在许多课本上，对相关论文的一篇概览是：D. Wood, "The Towers of Brahma and Hanoi revisited", *Journal of Recreational Math* 14 (1981-82), 17-24.

通过引入递归而消除栈的证明出现于：S. Brown, D. Gries and T. Szymanski, "Program schemes with pushdown stores", *SIAM Journal on Computing* 1(1972), 242-268.

参考前一章结尾列出的参考书目中有关递归的例子和应用。较早的包含了马的周游和八皇后问题的算法的书是：N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, Englewood Cliffs, N. J., 1976.

对 n-皇后问题的一般讨论，包括解的数目不能被限制在任何 n 的多项式的证明，参见 Igor Rivin, Ilan Vardi, and Paul Zimmermann, "The n-Queens Problem", *The American Mathematical Monthly* 101(7) (1994), 629-639.

关于递归的许多其他应用出现于如下这本书中：E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978, 626 pages.

这本书(第290～302页)包含有对游戏树和预测问题的更广泛的讨论和分析。递归的通用理论形成了一个研究主题，从理论方法方面可读的一篇文稿是：R. S. Bird, *Programs and Machines*, John Wiley, New York, 1976.

第 6 章　表和字符串

本章从受限表转向讨论更通用的表。在受限表中，修改仅发生在表的一端，如栈和队列；而在通用表中，插入、删除和检索则可能发生在表中任何位置。在检查这种表的说明和实现后，我们将研究称为字符串的字符表，设计一个简单的文本编辑器作为应用程序，并最终考虑数组链表的实现。

6.1　表的定义

我们开始定义称为表(list)的抽象数据类型。像栈和队列一样，表也有一个序列的元素作为它的数据值，然而与栈或队列不同的是，表允许改变序列中任意元素的操作。

定义　元素类型为 T 的表是 T 的元素的一个有限序列，加之下面的操作：

1. construct：构造表，使它为空。
2. empty：判断此表是否为空。
3. full：判断此表是否已满。
4. size：找出表的长度。
5. clear：将此表清空。
6. insert：在表中指定位置插入一个元素。
7. remove：删除表中指定位置的元素。
8. retrieve：检索表中指定位置的元素。
9. replace：替换表中指定位置的元素。
10. traverse：遍历此表，对表中每个元素执行给定的操作。

还有许多其他的可应用于元素序列的有用的操作，因此我们能利用不同的操作软件包来形成多种 ADT，这些相关的 ADT 中的任一个都能合理地以表的名义运行。但是，我们将注意力放在一种特定的表 ADT 上，它的操作给出了用表工作时引发的有代表性的思想和问题的样例。

* * * 标准模板库

标准模板库(STL)提供了一种相当不同的称为表的数据结构。STL list(STL 表)仅提供了在被认为是双向链的 List 实现中能有效地实现的那些操作，稍后我们将对其进行研究。特别地，STL 表不允许像我们的表操作中那样为插入、删除、检索以及替换操作提供对表中任意位置的随机访问。另一种称为 vector 的 STL 模板类提供了对数据值序列的某些随机操作。STL vector 与我们的 List ADT 有某些相似性，特别是它提供了我们称为顺序的 List 实现中能有效地实现的操作。这样，我们对 List ADT 的学习提供了对 STL 类 list 和 vector 的介绍。

6.1.1 方法说明

* * * 操作、信息隐藏和实现

在开始研究栈时，我们将栈的使用从这些操作的实际程序设计中分离出来，从而应用信息隐藏。在研究队列时，我们继续采用这一实践并很快发现在实现中可能有许多变化。对通用表，我们在访问和修改表的任意部分中的元素时有更大的灵活性和自由度。相比较栈和队列，信息隐藏原理对通用表来说更为重要，因此我们从列举希望可用的各种表方法的后置条件开始。

表在使用前需要构造函数：

* * * 构造函数

```
List::List();
后置条件：创建 List 并将其初始化为空。
```

下一个操作是取一个已存在的表并清空它：

* * * 重新初始化

```
void List::clear();
后置条件：删除所有的 List 元素，List 为空。
```

接下来是检查表状态的操作：

* * * 状态操作

```
bool List::empty() const;
后置条件：此函数根据 List 是否为空而返回 true 或 false。
```

```
bool List::full() const;
后置条件：此函数根据 List 是否已满而返回 true 或 false。
```

```
int List::size() const;
后置条件：此函数返回 List 中的元素个数。
```

现在来考虑对表中的元素进行访问的操作。就像我们以前对栈和队列的处理一样，假设每当必要的时候，我们的方法就会返回一个 Error_code 报告问题。我们将使用一种称为 List_entry 的泛型来表示表元素。

* * * 表中的位置

* * * 实现的独立性

为了找到表中的一个元素，我们使用整数给出它在表中的位置。通过给表中的位置编号，使得表中的第一个元素的位置为 0，第二个元素的位置为 1，以此类推。因此通过元素的

位置来定位它，表面上像是对一个数组进行索引，但其间却有重大的差别。首先，如果我们在特定位置插入一个元素，则其后继元素的位置需加 1。如果删除表中一个元素，则其后继元素的位置需减 1。此外，表的位置编号的定义没有考虑它的实现，对用一个数组实现的顺序表，此位置实际上会是数组中这个元素的下标，在链式实现中完全不使用数组和下标，但是我们仍然使用位置来查找链式实现的表元素。

现在我们给出访问表中单个元素的方法的准确说明。

```
Error_code List::insert(int position, const List_entry &x);
```
后置条件：若 List 不满且 $0 \leqslant$ position $\leqslant n$，这里 n 是 List 表中的元素数，则函数成功：先前在 position 位置上的任何元素和所有其他的后继元素将其位置编号加 1，并将 x 插入到 List 中的 position 处。
否则，此函数失败，并给出诊断错误码。

注意，insert 允许 position==n，因为将一个元素插入到表的最后一个元素之后是允许的。然而，下列方法要求 position<n，因为它们必须引用表中已存在的位置。

```
Error_code List::remove(int position, const List_entry &x);
```
后置条件：若 $0 \leqslant$ position $< n$，这里 n 是 List 表中的元素数，则函数成功：在 position 位置上的元素被从 List 中删除，并且所有的后继元素将其位置编号减 1，参数 x 记录了 List 中的原先 position 处的元素的副本。
否则，此函数失败，并给出诊断错误码。

```
Error_code List::retrieve(int position, const List_entry &x) const;
```
后置条件：若 $0 \leqslant$ position $< n$，这里 n 是 List 表中的元素数，则函数成功：在 position 位置上的元素被复制到 x 中，所有 List 的元素保持不变。
否则，此函数失败，并给出诊断错误码。

```
Error_code List::replace(int position, const List_entry &x);
```
后置条件：若 $0 \leqslant$ position $< n$，这里 n 是 List 表中的元素数，则函数成功：在 position 位置上的元素被 x 替换，所有 List 的元素保持不变。
否则，此函数失败，并给出诊断错误码。

* * * 遍历

遍历表以对表中元素执行某种任务的方法经常证明是有用的，特别是用于测试目的。客户程序使用这个遍历方法，指定表中各个元素要执行的某个动作，此动作逐个应用到表中的每个元素上。例如，客户程序有两个函数：void update(List_entry &x)和 void modify(List_entry &x)，及一个对象 List the_list，客户程序能使用命令 the_list. traverse(update)或 the_list. traverse(modify)在表中的每个元素上执行一个或另一个操作。如果客户程序想作为支架打印出表中所有元素，则它所需要的全部工作就是一条语句：the_list. traverse

(print);。这里的 void print(List_entry &x)是打印单个 List_entry 的函数。

* * * 指向函数的指针

在对方法 traverse 的这些调用中，客户程序仅提供函数的名字作为参数被处理。在 C++中，不带任何圆括号的函数名字被作为指向函数的指针。方法 traverse 的形参 visit 因此必须声明成一个指向函数的指针。此外，这个指针的声明必须包括的信息有：函数 * visit 拥有 void 返回类型和一个 List_entry 类型的引用参数。因此，我们得到方法 traverse 的如下说明：

```
void List::traverse(void(* visit)(List_entry &));
```
后置条件：在 List 的每个元素上执行函数 * visit 指定的动作，从位置 0 开始并依次执行。

与所有的参数一样，visit 只是一个形式名字，当使用 traverse 方法时要用一个指针来初始化它，visit 的所有参数亦是如此。表达式 * visit 代表用于遍历过程中处理表中每个元素的函数。

在下一节，我们将开始介绍实现问题。

练习 6.1

给定本节中描述的表方法，编写函数完成下面每个任务。确信为每个函数说明前置条件和后置条件。可以使用 List 和 List_entry 类型的局部变量，但不要编写依赖于实现选择的任何代码。要编写代码，在函数不能正常完成时进行检测和报告错误。

E1. Error_code insert_first(**const** List_entry &x, List &a_list)将元素 x 插入到 List a_list 的位置 0。

E2. Error_code remove_first(List_entry &x, List &a_list)删除 List a_list 的第一个元素，并将它复制到 x 中。

E3. Error_code insert_last(const List_entry &x, List &a_list)插入 x 作为 List a_list 的最后一个元素。

E4. Error_code remove_last(List_entry &x, List &a_list)删除 a_list 的最后一个元素，并将它复制到 x 中。

E5. Error_code median_list(List_entry &x, List &a_list) 如果 a_list 有奇数个元素，则将 List a_list 的中央元素复制到 x 中，否则将 a_list 的左中央元素复制到 x 中。

E6. Error_code interchange(**int** pos1, **int** pos2, List &a_list) 交换 List a_list 位置 pos1 和 pos2 上的元素。

E7. **void** reverse_traverse_list(List &a_list, **void** (* visit)(List_entry &))以逆序从尾到头遍历 List a_list。

E8. Error_code copy(List &dest, List &source)将所有元素从 source 复制到 dest，source 保持不变。你可以假设 dest 已经存在，但复制完成后 dest 中的任何元素将被废弃。

E9. Error_code join(List &list1, List &list2)将所有元素从 list1 复制到 list2 的尾部，list1 保持不变，list2 前面的所有元素亦保持不变。

E10. **void** reverse(List &a_list)反转 a_list 中所有元素的顺序。

E11. Error_code split(List &source, List &oddlist, list &evenlist)从 source 复制元素，使那些奇数编号的位置上的元素组成 oddlist，那些偶数编号的位置上的元素组成 evenlist。你可以假设 oddlist 和 evenlist 已经存在，但它们可能包含的任何元素将在复制后被废弃。

6.2　表的实现

至此，我们已说明了我们希望表 ADT 的操作如何进行，现在可以开始讨论在 C++中实现表的细节了。在前面对栈和队列的研究中，我们为两种实现编制程序，即使用数组的顺序实现和使用指针的链式实现。对表我们也有同样的划分，但我们会发现一些更重要的变化。

我们将自己的表作为通用模板类而不是普通的类来实现，因此先简单地复习一下模板的概念。

6.2.1　类模板

假设某客户程序需要 3 个表：一个整数表、一个字符表和一个实数表。那么我们迄今为止所设计的实现工具对它来说不够用，因为如果用 typedef 将类型 List_entry 设置成 int，char 或 double 其中之一，则就不能使用同样的 List 类再去设置具有另外两种类型的元素的表。因此需要设置一种泛化的表，这个表中的元素类型并未指定，但客户程序能够对它进行特别指明以便可以用 3 种元素来声明表。

* * * 模板

在 C++中，我们用模板(template)结构来实现这个目标，它允许我们使用泛化类型的对象来编写代码，这些代码经常是用来实现一个类。在模板代码中，使用参数指示这种泛化类型，并在以后客户程序使用模板代码时能够用一种实际的类型替代模板的参数。客户程序因此能通过用不同的实际类型替代模板的参数而从一个模板得到几种实际的代码段。

例如，我们将实现依赖于一个通用类型的参数的模板类 List，客户程序然后能使用此模板声明一个整数表，可以采用下面的声明形式：List<**int**> first_list;。另外，在同一程序中，这个客户程序也能用如下的声明创建一个字符表：List<**char**> second_list;。在这些声明语句中，我们的客户程序通过指定尖括号内的模板参数的值对模板进行自定义。

* * * 通用性

模板提供了一种新的创建普通数据结构的机制，使用模板而不是使用我们以前那种简单的通用处理所具有的一个优点是：客户程序在单个应用程序中可以对一种给定的数据结构模板做出不同的特化。例如我们前面声明的 first_list 和 second_list 有不同的类型，但却能共存于同一个客户程序中。在前面的通用处理形式中，正是缺乏这种灵活性，从而限制了我们在 4.5 节的多项式项目中对栈的选择。

通过模板得到的附加的通用性所产生的代价是更复杂的类说明和实现。对大多数部分，只需要在使用模板的代码前加上关键字 template 和模板参数的说明。因此后面的模板类 List 使用称为 List_entry 的泛化元素类型，通过向下面的说明中加入成员进行定义：

```
template<class List_entry>
class List {
  //  Add in member information for the class.
};
```

6.2.2 顺序实现

在顺序表实现中，是在一个有 max_list 个元素、类型为 List_entry 的数组中存入表数据。就像对顺序栈的处理一样，必须对表数据真正占用了数组中多少位置保持计数。因此必须定义一个类，使它具有我们的表 ADT 的所有方法和其他两个数据成员。

```
template<class List_entry>
class List {
public:
//  methods of the List ADT
  List();
  int size()const;
  bool full()const;
  bool empty()const;
  void clear();
  void traverse(void (*visit)(List_entry &));
  Error_code retrieve(int position, List_entry &x)const;
  Error_code replace(int position, const List_entry &x);
  Error_code remove(int position, List_entry &x);
  Error_code insert(int position, const List_entry &x);
protected:
//  data members for a contiguous list implementation
  int count;
  List_entry entry[max_list];
};
```

许多方法(list,clear,empty,size,retrieve)有非常简单的实现，然而，这些函数却依赖于模板参数 List_entry，因此它们也必须作为模板实现。例如，方法 size 能用下面的函数模板编写：

```
template<class List_entry>
int List<List_entry>::size()const
/* Post: The function returns the number of entries in the List. */
{
  return count;
}
```

我们将其他的简单方法留作练习，而将注意力集中在那些访问表中数据的方法。为了向表中加入元素，必须在数组中移动元素以便为新插入的元素腾出空间，结果得到的函数模板为：

```
template<class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/* Post: If the List is not full and 0 <= position <= n, where n is the number of entries in
        the List, the function succeeds: Any entry formerly at position and all later
        entries have their position numbers increased by 1 and x is inserted at position
```

```
        of the List.
        Else: The function fails with a diagnostic error code. */
{
  if (full())
    return overflow;
  if (position<0 || position>count)
    return range_error;
  for(int i=count - 1; i>=position; i--)
    entry[i+1]=entry[i];
  entry[position]=x;
  count++;
  return success;
}
```

这个函数需要做多少工作？如果我们在表尾插入一个元素，则此函数只执行很少的常量级的命令；如果是另一种极端情况，我们在表的开始处插入元素，则函数必须移动表中的每个元素以腾出空间。因此，如果表比较长的话，它将做大量工作。在平均情况下，我们可以假设所有可能的插入是同等可能，则此函数要移动表中一半的元素。因此此函数的工作量与表长 n 近似成比例。

类似地，删除也必须移动表中的元素以填补被删元素留下的空位，因此删除也要求与元素个数 n 近似成比例的时间。另一方面，剩余的大多数操作不使用任何循环，因而可以在常量时间内完成工作。总之，

在处理有 n 个元素的顺序表时：

- insert 和 remove 要求与 n 近似成比例的时间。
- List，clear，empty，full，size，replace 和 retrieve 在常量时间内操作。

我们在讨论中没有包括 traverse，因为它的时间依赖于其参数 visit 所用的时间。遍历的实现必须在表的所有 n 个元素上都走一遍，因此我们不能期望它的时间需求会少于 n 的比例。然而对具有固定 visit 的遍历，其时间需求与 n 近似成比例。

```
template<class List_entry>
void List<List_entry>::traverse(void (*visit)(List_entry &))
/* Post: The action specified by function(*visit) has been performed on every entry of
         the List, beginning at position 0 and doing each in turn. */
{
  for(int i=0; i<count; i++)
    (*visit)(entry[i]);
}
```

6.2.3　简单的链式实现

1. 声明

我们从结点的声明开始介绍表的链式实现，这里的结点类似于我们在链栈和链式队列中使用的那些结点，但我们现在让它们依赖于模板参数。

```
template<class Node_entry>
```

```
struct Node {
//  data members
  Node_entry entry;
  Node<Node_entry>*next;
//  constructors
  Node();
  Node(Node_entry, Node<Node_entry>*link=NULL);
};
```

我们包含了两个构造函数，对它们的选择依赖于是否要初始化 Node 的内容。这些构造函数的实现与在 4.1.3 节中使用的那些链式结点几乎一样。一旦我们定义完 Node 结构，我们就可以通过填充下面的框架给出链表的定义。

```
template<class List_entry>
class List {
public:
//  Specifications for the methods of the list ADT go here.
//  The following methods replace compiler-generated defaults.
  ~ List();
  List(const List<List_entry>&copy);
  void operator=(const List<List_entry>&copy);
protected:
//  Data members for the linked list implementation now follow.
  int count;
  Node<List_entry>*head;
//  The following auxiliary function is used to locate list positions
  Node<List_entry>*set_position(int position)const;
};
```

在此定义中，我们省略了方法原型，因为它们与顺序实现中所用的那些方法是一样的。与受保护的数据成员一样，我们包含了受保护的函数 set_position，它将在这些方法的实现中证明是有用的。

2. 例子

为了详细说明需要链表执行的某些操作，我们先花一点时间考虑一下编辑文本的问题，并假设每个结点既拥有指向下个结点的链，也包含一个单词。图 6.1 (a)中所示的是句子“Stacks are Lists”，如果我们将单词“simple”插入到单词“Lists”之前，我们得到(b)中的表。接下来我们打算用“structures”替换“Lists”并插入 3 个结点“but important data”，可以得到(c)。之后，我们决定删除“simple but”，因此得到表(d)。最后，我们遍历这个表以打印出它的内容。

3. 查找表的位置

表的几个方法要用到一个名为 set_position 的函数，它将一个位置(即表中的一个整数下标)作为参数，并返回指向表中相应结点的指针。

我们要将 set_position 的可见性声明为受保护的，这是因为 set_position 返回指向 List 中一个 Node 结点的指针，并因而能访问这个结点，因此能访问 set_position 的任何客户程序将能够访问相应的 List 中的所有数据。为了维护一个封装的数据结构，必须限制 set_

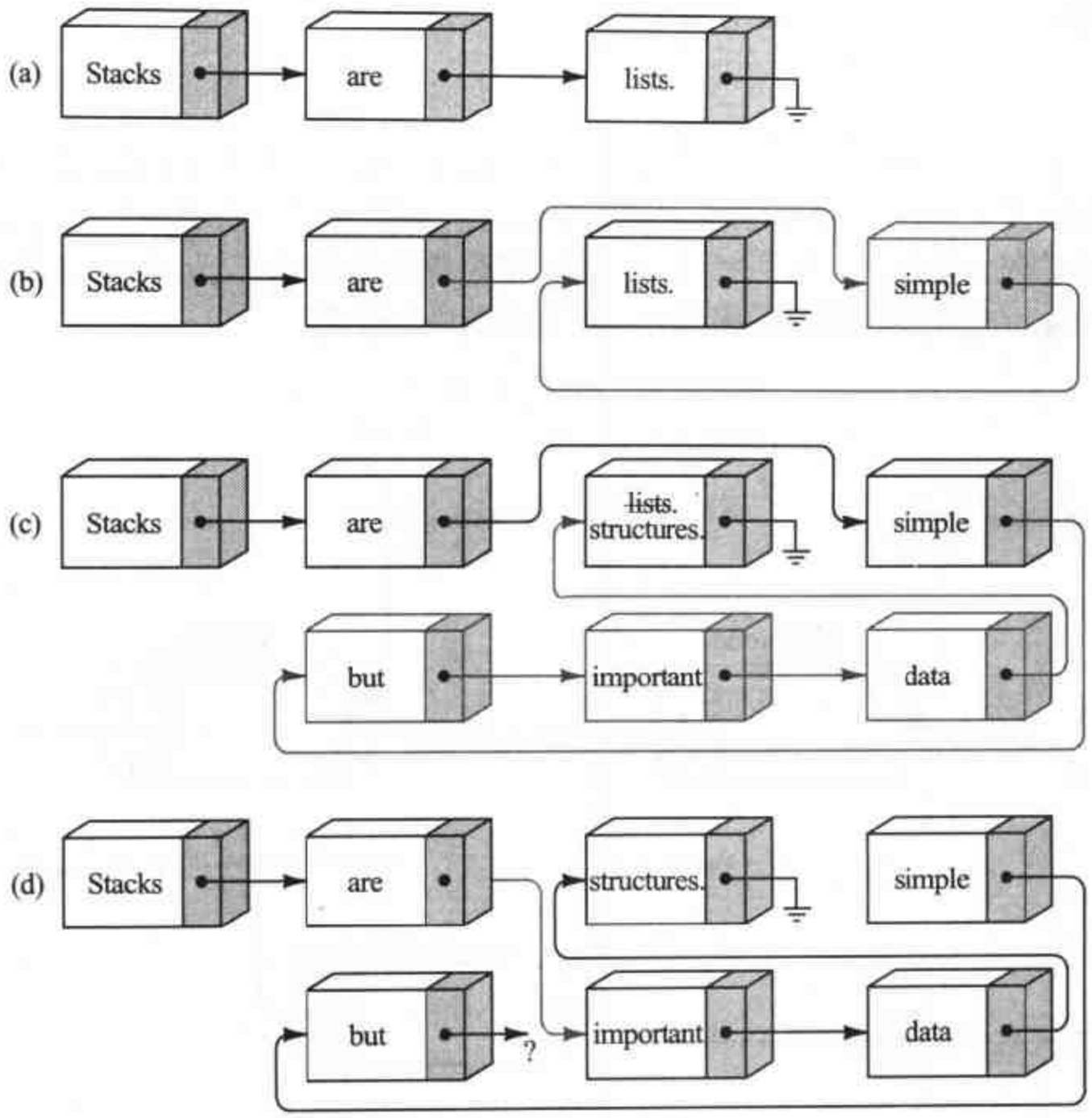


图 6.1　链表上的操作

position 的可见性，通过赋予它受保护的可见性，可以确保它只能作为构造 List 中其他方法时可用的工具。

从概念上看，构造 set_position 的最简单的方式是从 List 的起始处开始并遍历它，直至所需要的结点。

```
template<class List_entry>
Node<List_entry>* List<List_entry>::set_position(int position)const
/* Pre:  position is a valid position in the List; 0≤position<count.
   Post: Returns a pointer to the Node inposition.*/
{
   Node<List_entry>*q=head;
   for(int i=0; i<position; i++)q=q->next;
   return q;
}
```

由于我们准确地控制哪些函数可以使用 set_position，因此就没有必要再包含错误检查，替之以利用前置条件。实际上调用 set_position 的函数会包含其自己的错误检查，因此在 set_position 中再次重复这个过程就降低了效率。

如果所有结点都是同等可能，则平均起来，set_position 函数在查找给定的位置时，必须

从头到尾移动 List 的半程，因此，平均来说，它的时间需求与 List 的长度 n 近似成比例。

4. 插入

接下来我们考虑向链表 List 中插入新元素的问题。如图 6.2 所示，如果有一个新元素要插入到链式 List 的中间位置，那么我们创建指向新元素插入位置的前趋(preceding)和后继(following)结点的指针。如果设 new_node 指向待插入的结点，privious 指向其前趋结点，following 指向其后继结点，则此操作由下面两条语句组成：

```
new_node->next=following;
previous->next=new_node;
```

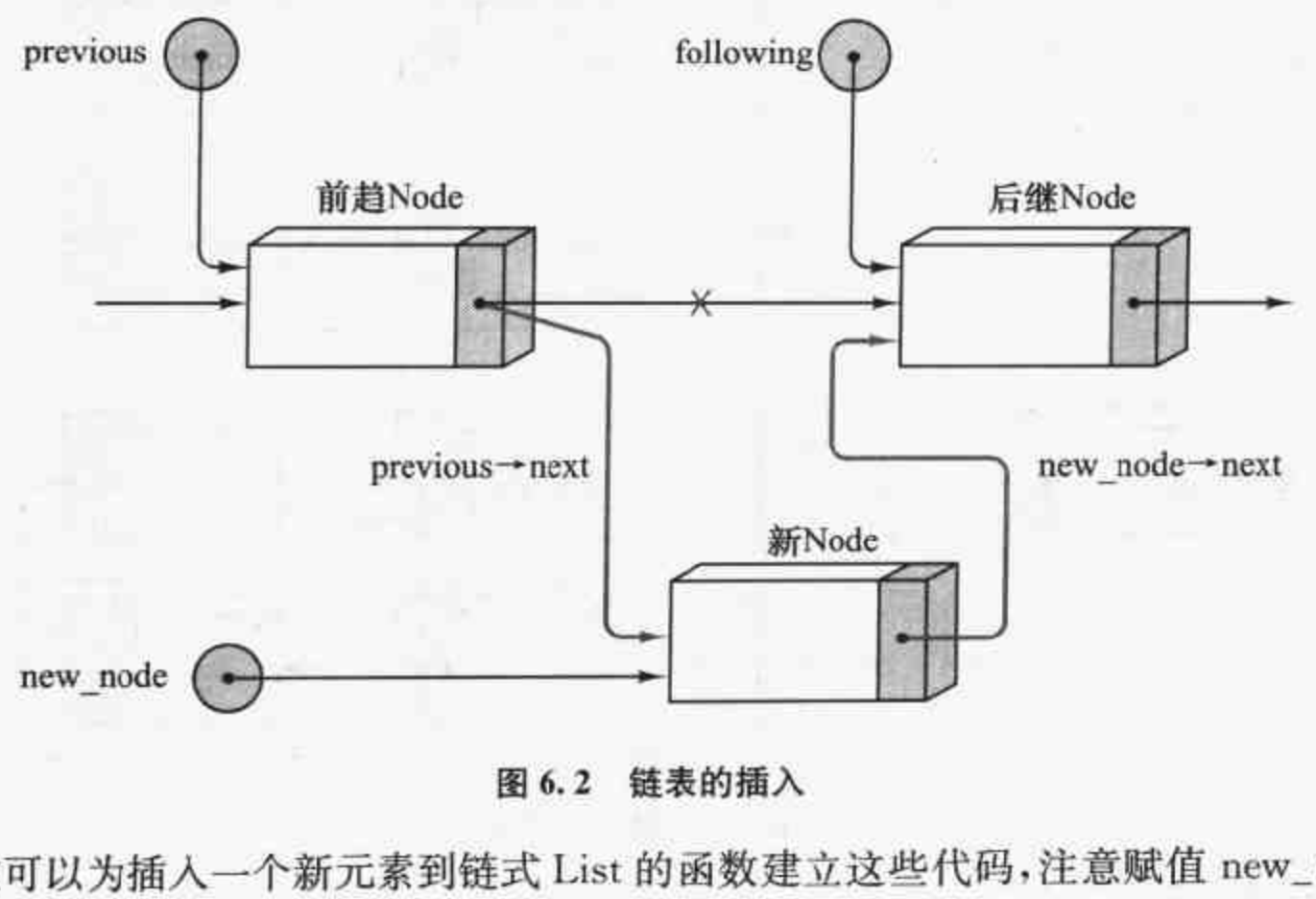


图 6.2 链表的插入

现在可以为插入一个新元素到链式 List 的函数建立这些代码，注意赋值 new_node->next=following 是由初始化 new->node 的构造函数来执行。在 List 起始处的插入必须作为特殊情况处理，因为新的元素不会作为任何其他元素的后继。

```
template<class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/* Post: If the List is not full and 0≤position≤n, where n is the number of entries in
         the List, the function succeeds: Any entry formerly at position and all later
         entries have their position numbers increased by 1, and x is inserted at
         position of the List.
         Else: The function fails with a diagnostic error code. */
{
  if (position<0||position>count)
    return range error;
  Node<List_entry>*new_node,*previous,*following;
  if (position>0){
    previous=set_position(position-1);
    following=previous->next;
  }
  else following=head;
  new_node=new Node<List_entry>(x, following);
  if (new_node==NULL)
```

```
    return overflow;
  if (position==0)
    head=new_node;
  else
    previous->next=new_node;
  count++;
  return success;
}
```

除了对 set_position 的调用以外，insert 执行的步骤不依赖于 List 的长度，因此它的操作时间与 set_position 一样，同 List 的长度 n 近似成比例。

5. 其他操作

链表的其他操作都将留作练习。那些访问 List 中特定位置的操作都需要用到函数 set_position，有时是当前的位置，有时是像 insert 函数中那样取前趋的位置。除了 clear（和 traverse）这两个函数需要作用于 List 中的所有元素以外，所有其他函数如果不考虑 set_position 中的步骤的话，都最多执行常数个步骤。我们因此得到以下结论：

在处理 n 个元素的链表时：
- clear，insert，remove，retrieve 和 replace 要求的时间与 n 近似成比例。
- List，empty，full 和 size 在常量时间内操作。

我们在讨论中再次没有包括 traverse，因为它的时间依赖于其参数 visit 所用的时间，而这个时间我们一般并不知道。但是，同前面一样，对固定的参数 visit，traverse 所需要的时间与 n 近似成比例。

6.2.4　变化：保留当前位置

许多应用程序是按序处理表中的元素，将一个元素移到下一个位置；许多其他的应用程序要几次引用同一个元素，在移动到另一个元素之前做 retrieve 或 replace 操作。对所有这些应用，我们当前的链表实现显得效率低下，因为访问表中某个元素的每个操作都从表的起始处开始查找，直至想要的位置。如果能够记住表中上一次使用的位置，并且下一个操作引用同一个或后一个位置，则从上次用过的位置开始查找将会更有效率。

然而注意，记住上次用过的位置不会加速使用表的每个应用程序，例如，如果某些程序从链表的尾部开始，以逆序访问链表中的元素，则每次访问都需要从表的起始处开始查找，因为链仅是单向的，使得记住上次用过的位置无助于查找它的前趋结点。

方法 retrieve 会产生一个问题，这个方法定义成一个常量方法，但是它的实现却需要改变 List 上次用过的位置。我们认为虽然这个操作改变了 List 对象的某些数据成员，但它没有改变代表这个对象的真实值的元素的顺序。为了确保 C++编译器承认，必须用存储修改符 mutable 定义一个数据成员以记录 List 的上次用过的位置。关键词 mutable 新近才加到 C++中，并且迄今在此语言的所有实现中还不可用。类的 mutable 数据成员能被修改，即使是常量方法也能修改它。

通过将方法说明加入下面的框架中可以得到表的增大的定义：

```
template<class List_entry>
class List {
public:
//  Add specifications for the methods of the list ADT.
//  Add methods to replace the compiler-generated defaults.
protected:
//  Data members for the linked-list implementation with
//  current_position follow:
  int count;
  mutable int current_position;
  Node<List_entry>*head;
  mutable Node<List_entry>*current;
//  Auxiliary function to locate list positions follows:
  void set_position(int position)const;
};
```

注意虽然我们在先前的类定义中加入了额外的成员，所有的新成员都具有受保护的可见性，这意味着从客户程序的观点，这个类看起来确实像我们先前的实现。

可以重新编写 set_position 来使用和修改这个类的新的数据成员，当前的位置现在是类 List 的一个成员，因此不再需要 set_position 返回一个指针，相反，这个函数只需简单地重置 List 中的指针 current 即可。

```
template<class List_entry>
void List<List_entry>::set_position(int position)const
/* Pre:  position is a valid position in the List: 0<=position<count.
  Post: The current Node pointer references the Node at position.*/
{
  if (position<current_position){ //  must start over at head of list
     current_position=0;
     current=head;
  }
  for(; current_position !=position; current_position++)
    current=current->next;
}
```

注意，对同一位置的重复引用，if 语句体和 for 语句体都不会执行，因此，此函数几乎不花什么时间。如果仅向前移动一个位置，则 for 循环体将仅执行一次，因此此函数还是非常快。另一方面，当必须在 List 中从后向前移动时，此函数将以前面的实现中所用的 set_position 版本的同样的方式操作。

利用 set_position 的这个修订版本，我们现在能够修订链表的实现以改进它的效率，对各种方法要做出的改动很少，因此将留作练习。

6.2.5 双向链表

链表的某些应用要求在表中频繁地进行前后移动。在上一节中，我们从表头开始遍历，直到发现所要的结点，从而解决向后移动的问题，但这种解决方案通常难以令人满意。不仅它的程序设计困难，而且程序的运行时间将依赖于表的长度，而这个表可能会很长。

可以采用几种策略来克服在链表中查找某个给定结点的前趋结点的问题，我们在本节将研究最简单的、并且在大多数情况下最灵活而且令人满意的一种策略。

1. 双向链表的声明

* * * 双向链表

双向链表的思想如图 6.3 所示，它对每个结点保留两个链，分别指向相反的方向。因此沿着合适的链而行，可以在链表中按任一方向移动，而且同样简单，我们称这种表为双向链表。

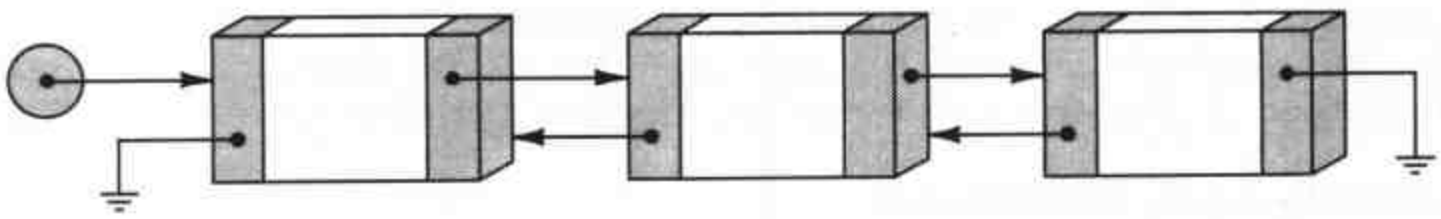


图 6.3　双向链表

在双向链表中，结点的定义变成：

```
template<class Node_entry>
struct Node {
//  data members
  Node_entry entry;
  Node<Node_entry>*next;
  Node<Node_entry>*back;
//  constructors
  Node();
  Node(Node_entry, Node<Node_entry>*link_back=NULL,
                   Node<Node_entry>*link_next=NULL);
};
```

Node 构造函数的实现只是对 4.1.3 节中单向链表构造函数稍做修改，因此我们马上就可以直接得到双向链表类的框架定义。

```
template<class List_entry>
class List {
public:
//  Add specifications for methods of the list ADT.
//  Add methods to replace compiler generated defaults.
protected:
//  Data members for the doubly-linked list implementation follow:
  int count;
  mutable int current_position;
  mutable Node<List_entry>*current;
//  The auxiliary function to locate list positions follows:
  void set_position(int position)const;
};
```

在这种实现中，可能只在 List 中保留一个指针，同时在 List 中按任一方向移动，因此在声明中，仅保留指向 List 当前结点的指针。我们甚至不必保留指向 List 的头或尾的指针，因为同其他指针一样，它们也从任意给定的结点通过前后查找来得到。

2. 双向链表的方法

采用双向链表，表的双向检索、特定位置的查找、任意位置的插入和删除都能够毫无困难地实现。在表中进行修改的一些方法会比简单的链表更长，这是因为当在表中插入或删除一个元素时，它必须更新前后两个方向的链。

首先，为了找到表中某个特定位置，仅需要确定是从初始位置向前还是向后移动。然后再在此表上进行局部遍历，直至想要的位置。结果函数为：

```
template<class List_entry>
void List<List_entry>::set_position(int position) const
/* Pre:  position is a valid position in the List: 0≤position<count.
   Post: The current Node pointer references the Node atposition.*/
{
  if (current_position<=position)
    for(; current_position !=position; current_position++)
      current=current->next;
  else
    for(; current_position !=position; current_position--)
      current=current->back;
}
```

给出这个函数，现在就可以编写插入方法，此方法由于需要调整多个链而会比单向链的插入方法稍长一些。此函数的操作如图 6.4 所示。

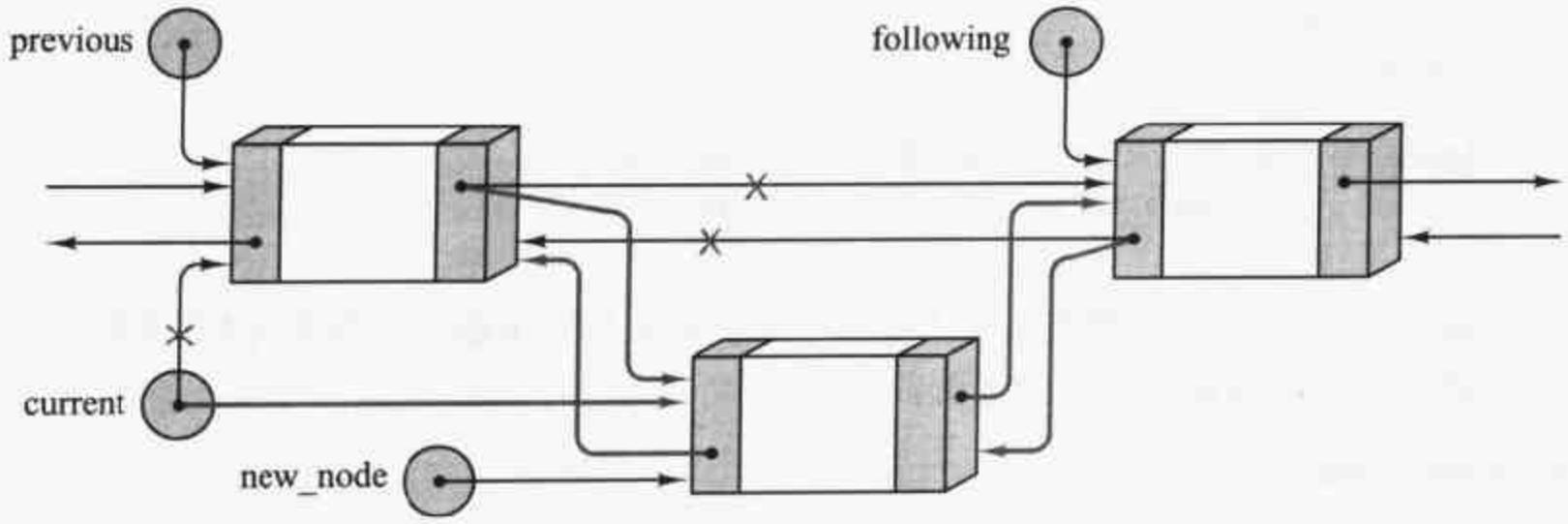


图 6.4　双向链表的插入

当在 List 的端点处插入或者向空表中插入时必须特别小心。

```
template<class List_entry>
Error_code List<List_entry>::insert(int position, const List_entry &x)
/* Post: If the List is not full and 0≤ position≤n, where n is the number of entries in
         the List, the function succeeds: Any entry formerly at position and all later
         entries have their position numbers increased by 1 and x is inserted at
         position of the List.
         Else: the function fails with a diagnostic error code. */
{
  Node<List_entry>*new_node,*following,*preceding;
  if(position<0||position>count)return range_error;
  if (position==0){
    if(count==0)following=NULL;
```

```
    else {
      set_position(0);
      following=current;
    }
    preceding=NULL;
  }
  else {
    set_position(position-1);
    preceding=current;
    following=preceding->next;
  }
  new_node=new Node<List_entry>(x, preceding, following);
  if(new_node==Null)return overflow
  if(preceding!=NULL)preceding->next=new_node;
  if(following !=NULL)following->back=new_node;
  current=new_node;
  current_position=position;
  count++;
  return success;
}
```

当然，双向链表的代价是每个 Node 的第二个链所要求的额外空间。然而，对大多数应用程序来说，每个 Node 中的信息成员(即元素)所需要的空间量远远多于一个链所需要的空间，因此每个 Node 中的第二个链并未明显地增加 List 需要的存储空间总量。

6.2.6　实现间的比较

由于我们已了解了一些对链表及其结构和实现中的变化进行操作的算法，现在就让我们稍做停顿，先来评价一下表的链式和顺序实现的相对优点。

＊＊＊优点

＊＊＊上溢

链表最主要的优点在于动态存储的灵活性，除非计算机内存耗尽，否则不会出现上溢。特别是当单个元素很大的时候，在为其他需求保留足够空间的同时，一般很难确定数组所需要的顺序静态存储的数量。而采用动态分配的话，就不必事先做出这样的决定。

＊＊＊修改

在链表的中间进行修改比顺序表更快，特别是插入和删除。如果结构较大的话，那么修改几个指针的值比将结构本身从一个地方复制到另一个地方要快得多。

＊＊＊缺点

链表的第一个缺点是链本身要占用空间，这些空间否则的话可以供另外的数据使用。在大多数系统中，指针需要与整型数同样的存储量(一个字)，因此整数表在链式存储中所需要的空间会是顺序存储中所需要的空间的两倍。

＊＊＊空间利用

另一方面，在许多实际应用中，表中的结点会相当大，其全部数据成员可能会占用几百个字。如果每个结点包含 100 个字的数据，则使用链式存储会增加百分之一的内存需求，这是一个可以忽略不计的量。实际上，如果需要为存放顺序表的数组分配额外的空间以考虑

辅助插入，则链式存储总共可能要求的空间会更少。如果每个元素占用 100 个字，则当所有数组超过 99%的容量都被填满时，顺序存储才会节省空间。

* * * 随机访问

* * * 程序设计

链表的主要缺点是不适合于随机访问。采用顺序表，客户程序都能尽可能快地指向表中的任何位置。而采用链表，它可能必须遍历很长的路径后才到达想要的结点。对链式存储中单个结点的访问可能平均稍微多花一点计算机时间，因为它首先必须获得指针，然后到达相应的地址。然而，最后一个因素通常并不重要。类似地，你可能在刚开始会发现操作链表的方法需要在程序设计上花费更多的努力，但一旦熟练了，这种差异也就消失了。

因此，我们总结如下：

顺序存储通常更可取：

- 当个别看起来，元素非常小时；
- 当编写程序时就已知表的长度时；
- 当除了在表尾外只需要很少的插入或删除时；
- 当随机访问显得很重要时。

链式存储证明更优：

- 当元素较大时；
- 当事先不知道表的长度时；
- 当在插入、删除和重排元素中需要灵活性时。

最后，为了帮助在数据结构和实现的许多种可能的变化中选择一种，程序设计者应当考虑在表上实际需要执行哪些操作，并且这些操作中哪些最为重要。存在引用的局部性吗？即如果一个元素被访问，它下次还可能接着被访问吗？元素是否要按顺序处理？如果是，则将上次用过的位置作为表数据结构的一部分来维护就是值得的。有必要在表中双向移动吗？如果是，则双向链表可以证明其优越性。

练习 6.2

E1. 编写 C++函数实现 List 的顺序实现中剩余的操作，如下：

(1)构造函数 List　(5)replace

(2)clear　(6)retrieve

(3)empty　(7)remove

(4)full

E2. 编写 C++函数，用单链和双向链 Node 对象实现构造函数(两种形式)。

E3. 编写 C++函数，为表的(第一种)简单的链式实现完成下列操作：

(1)构造函数 List　(7)empty

(2)拷贝构造函数　(8)full

(3)重载赋值运算符　(9)replace

(4)析构函数~List　　(10)retrieve
(5)clear　　(11)remove
(6)size　　(12)traverse

E4. 为简单链表的(第二种)实现编写 remove 方法,它记住上次用过的位置。

E5. 指出下列哪些函数对双向链表(如本节中的实现)和简单链表都是一样的。对那些不同的函数,则为双向链表编写一个新的版本。确信每个函数符合 6.1 节中给出的说明。

(1)构造函数 List　　(8)full
(2)拷贝构造函数　　(9)replace
(3)重载赋值运算符　　(10)insert
(4)析构函数~List　　(11)retrieve
(5)clear　　(12)remove
(6)size　　(13)traverse
(7)empty

程序设计项目 6.2

P1. 准备文件集,包含顺序表的声明和用于表处理的所有函数。

P2. 基于 3.4 节中的演示程序,为通用表编写一个菜单驱动的演示程序。表的元素应当为字符。使用项目 P1 中设计的通用表的函数和声明。

P3. 创建文件集,包含处理链表的声明和函数。
(1)使用第一种实现中的简单的链表。
(2)使用简单的链表,指针指向上一次使用过的位置。
(3)使用本节中实现的双向链表。

P4. 在项目 P2 的菜单驱动的演示程序中,用支持链表(来自项目 P3)的声明和函数构成的文件集替换支持顺序表(项目 P1)的文件集,如果你能够仔细设计这些声明和函数的话,那么不用对程序做进一步的修改,程序就可以正确地运行。

P5. (1)修改双向链表的实现,使得它维持指向表中首个结点和最后结点的指针,同时还有指向上次用过的位置的指针。
(2)用项目 P2 的菜单驱动的演示程序来使用这种实现,从而测试它是正确的。
(3)讨论双向链表的这种变化与课本中的双向链表之间的优缺点。

P6. (1)编写一个对任意大的整数做加、减、乘、除法的程序,要将每个整数都表达成它的数字的表。由于整数可随意大,因此需要使用链表以防止溢出的可能性。在表中向后移动对某些操作是有用的,因此双向链表较合适。乘法和除法可以简单地以重复的加法和减法来实现。
(2)重写乘法,使它不是基于重复的加法,而是基于标准的乘法,其中将第一个被乘数乘以第二个乘数中的每个数字,然后相加。
(3)重写除法操作,使它不是基于重复的减法,而是基于长除法。有必要编写一个附加的函数,判断被除数在绝对值上是否大于除数。

6.3 字符串

* * * 定义

本节要实现一个类来表示字符串。字符串(string)定义为一个字符序列,字符串的例子是"This is a string"或者"Name?",这里的引号(" ")不是字符串的一部分,空串用""表示。由于字符串存储数据(字符)序列,因此字符串 ADT 是一种表,然而由于通常应用于字符串的操作与我们的表 ADT 的操作显著不同,因此我们不会基于任何早先的表结构来实现字符串。

我们从复习 C++语言提供的字符串处理能力开始。

6.3.1 C++中的字符串

C++语言提供了两种字符串实现,其中较原始的一种只是字符串的 C 语言实现。与 C 语言的其他部分一样,它在 C++的所有实现中可用,我们将这种实现提供的字符串对象归为 C-串(C-string)。C-串反映了 C 语言的强大和弱点:它们广泛可用,十分有效率,但易于滥用而招致灾难性的后果。C-串必须符合一个约定集,我们现在复习一下这个约定集。

* * * 约定

每个 C-串有 char * 类型,因此,一个 C-串引用了内存中的一个地址,被引用的地址是存储了组成此字符串的字符的顺序字节集的开头,此字符串所占用的存储必须以特殊字符值'\0'结束。编译器无法实施任何这些约定,但对这些规则的任何偏离可能导致运行时崩溃。换言之,C-串的对象是未封装的。

标准头文件<cstring>包含操作 C-串的函数库。在较老的 C++编译器中,这个头文件有时称为<string.h>。这些库函数方便、高效,并且表达了我们希望使用的几乎每种字符串操作。例如,假设 s 和 t 是 C-串,则操作 strlen(s)返回 s 的长度,strcmp(s, t)展示 s 和 t 的字典顺序,strstr(s, t)返回一个指向字符串 t 在 s 中首次出现位置的指针。此外,在 C++中,重载了输出运算符<<以应用于 C-串,因此简单的指令 cout<<s 打印出字符串 s。

* * * 缺点

虽然 C-串的实现有许多优秀的特征,但它也有一些严重的缺点。实际上它恰恰遭受我们在 4.3 节研究链式数据结构时确定的问题,客户程序很容易为字符串数据创建垃圾或别名。例如在图 6.5 中,我们图示说明 C-串的赋值 s=t 如何导致这两个问题。

另一个问题经常出现在使用 C-串的应用程序中。未经初始化的 C-串应当存储 NULL 值,但是许多字符串的库函数在遇到 NULL 字符串对象时会失败(运行时崩溃)。例如下面的语句:

```
char*x=NULL;
cout<<strlen(x);
```

这两条语句能被编译器接受,但对 C-串库的许多实现来说,它们将产生致命的运行时错误。因此,客户程序的代码需要仔细测试 C-串函数的前置条件。

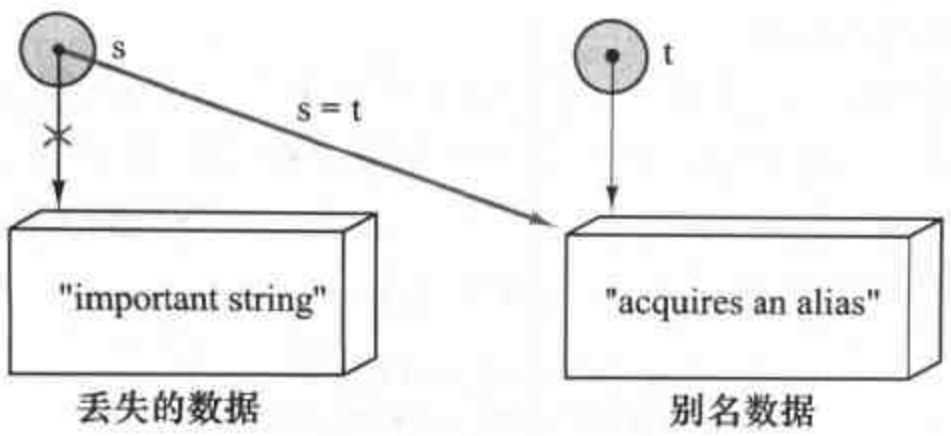


图 6.5　C-串对象的不安全性

在 C++中，容易使用封装来将 C-串嵌入更加安全的基于类的字符串实现中。实际上标准模板库在头文件<string>中包含了一种安全的字符串实现，这个库实现了一个称为 std::string 的类，它方便、安全且有效。由于这个库没有包含在较老的 C++编译器中，因此我们将设计自己的安全的 String ADT，使用封装和面向对象的技术来克服 C-串中发现的问题。

6.3.2　字符串的实现

＊＊＊ String 类

为创建更安全的字符串实现，需要将 C-串的表示嵌成 String 类的一个成员，将字符串的长度作为我们的类中的第二个数据成员也非常方便。此外，通过包含重载的赋值运算符、拷贝构造函数、析构函数和构造函数，我们的 String 类能够避免别名、垃圾创建和未经初始化的对象等问题。

＊＊＊比较运算符

对后面的应用，如果能够应用比较运算符<，>，<=，>=，==，!=来确定两个字符串之间的字典顺序关系，则会显得特别方便。因此，String 类将包含重载的比较运算符。

＊＊＊String 构造函数

我们也要为 String 类装备一个构造函数，它使用一个类型为 char＊的参数，此构造函数提供了从 C-串对象到 String 对象的方便的转换器，这个转换器能够以如下的代码被显式地调用：String s("some_string");。在此语句中，通过转换 C-串"some_string"来构造 String s。

无论何时当客户程序的代码需要从类型 char＊到 String 的类型转换时，我们的构造函数也能由编译器隐式地调用。例如，在运行下面的语句时，构造函数被调用：

```
String s;
s="some_string";
```

为翻译第二条语句，C++编译器首先调用我们的新构造函数将"some_string"转换成一个临时的 String 对象，然后调用重载的 String 赋值运算符将这个临时的字符串复制到 s 中，最后它为这个临时的 String 调用析构函数。

有一个类似的构造函数也很有用，它将一个字符型的 List 转换成一个 String。例如当从用户那里读取一个 String，最方便的方法是将字符读到一个链表中，一旦读取这个链表，就能应用我们的构造函数将此链表转换成一个 String。

* * * 从 String 到 C-串的转换

最后，如果能够将 String 对象转换成相应的 C-串对象也会有用。例如，这种转换将许多 C-串库函数应用于 String 数据。我们将遵循模板库的例子，并将这个转换实现为一个称为 c_str()的 String 方法，这个方法将返回一个 const char * 类型的值，这个值是指向表示此 String 的字符数据的指针。方法 c_str()可如下使用：

```
String s="some_string";
const char*new_s =s.c_str();
```

让方法 c_str()返回一个常量字符的 C-串是重要的，如果考虑 new_s 字符串所占用的计算机内存，我们能够认识到这种谨慎的必要性，此内存必然是由类 String 分配的。而且一旦分配，此内存必须由客户程序代码或由类 String 维护并最终删除。我们认为类 String 应该接受这个责任，因为这样允许我们为转换函数编写高效的实现，并且将避免客户程序忘记删除由 String 类创建的 C-串的可能性。然而，为此必须付出的代价是客户程序不应使用返回的指针去改变所引用的字符数据。因此，我们的转换返回一个常量 C-串。

我们描述的几个特征结合起来提供了非常灵活、强大并且安全的字符串处理。我们自己的 String 类是一种完全封装的 ADT，但它提供了到 C-串和字符表的完整的接口。

现在可以得到下面的类说明：

```
class String {
public:                    //  methods of the string ADT
  String();
  ~String();
  String(const String &copy); //  copy constructor
  String(const char*copy);    //  conversion from C-string
  String (List<char>&copy);   //  conversion from list
  void operator=(const String &copy);
  const char*c_str()const;    //  conversion to C-style string
protected:
  char*entries;
  int length;
};
bool operator==(const String &first, const String &second);
bool operator> (const String &first, const String &second);
bool operator< (const String &first, const String &second);
bool operator>=(const String &first, const String &second);
bool operator<=(const String &first, const String &second);
bool operator!=(const String &first, const String &second);
```

* * * 构造函数

将 C-串和 List 数据转换成 String 对象的构造函数如下实现：

```
String::String (const char*in_string)
/* Pre:  The pointer in_string references a C-string.
  Post: The String is initialized by the C-string in_string.*/
{
  length=strlen(in_string);
  entries=new char[length+1];
  strcpy(entries, in_string);
```

```
}
String::String (List<char>&in_list)
/* Post: The String is initialized by the character list in_list.*/
{
  length=in_list.size();
  entries=new char[length+1];
  for(int i=0; i<length; i++)in_list.retrieve(i, entries[i]);
  entries[length]='\0';
}
```

我们选择实现转换方法 c_str()，它将 String 转换成类型 const char*，如下所示：

```
const char*String::c_str()const
/* Post: A pointer to a legal C-string object matching the String is returned.*/
{
  return(const char*)entries;
}
```

*** 效率的折衷

这个实现并非完全令人满意，因为它提供了到内部 String 数据的访问。但是就像我们将要解释的，其他的实现和策略也有问题，而这种实现具有最高效率的优点。我们注意到标准的模板库在 c_str()的实现中亦做出了类似的折衷。

*** 别名问题

方法 c_str()返回一个客户程序可读不可改的指向字符数组的指针；在此情况中，我们选择返回对 String 的 C-串数据成员的访问。我们使用一个转换以确保所返回的指针引用常量 C-串，但不负责任的客户程序能够类似地抛弃所返回的 C-串的不变性，并因而打破类的封装。一个更严重的问题是我们的函数造成的别名问题，这意味着我们的类 String 或 STL 类 string 的客户程序只能在应用这个方法之后立即使用 c_str()的结果，否则，即使是负责任的客户程序也能碰到下面的代码所展现的问题。

```
String s="abc";
const char*new_string=s.c_str();
s="def";
cout<<new_string;
```

语句 s="def"; 导致前面的 String 和 C-串数据的删除，因而最后的语句有不可预测的结果。

*** 可选的实现

c_str()的一种可供选择的实现是为 String 数据的副本分配动态内存并将字符复制到此存储区，指向动态 C-串的指针被作为函数的返回值转交给客户程序。这种选择明显效率较低，特别是在转换长的字符串时。而且，它有另外一个严重的缺点：它非常可能导致垃圾的创建。客户程序必须记住在使用 C-串后删除它。例如下面的语句对我们前面的 c_str()方法的实现不会带来什么问题，但如果采用可供选择的实现，那么它会产生一些垃圾。

```
String s="some very long string";
cout<<s.c_str();        //  creates garbage from a temporary C-String object
```

*** 重载的比较运算符

最后，我们讨论重载的比较运算符。下面的重载运算符==的实现很短而且非常有效，

这正是因为我们的方法 String::c_str()的方便性和效率。

```
bool operator==(const String &first, const String &second)
/* Post: Return true if the String first agrees with String second. Else: Return false.
        */
{
return strcmp(first.c_str(), second.c_str())==0;
}
```

我们在重载运算符--=时所使用的语法类似于 4.3.2 节中实现的重载运算符中所使用的语法。其他重载的比较运算符有几乎相同的实现。

6.3.3 更多的字符串操作

我们现在设计许多操作 String 对象的函数。由于用户可能知道 C-串的库中的方法，因此我们将为 String 对象创建一个类似的库。

在许多情况下，C-串函数可以直接应用于转换后的 String 对象，例如，不需额外的程序设计工作，我们能合法地编写出：

```
String s="some_string";
cout<<s.c_str()<<endl;
cout<<strlen(s.c_str())<<endl;
```

* * * 重载的函数

对诸如 strcpy 这样不改变字符串的自变量的 C-串函数，我们需要编写以 String 参数而非 char * 参数操作的重载的版本。就如我们已经提到的，在 C++中：当在一个程序的单个范围内包含一个函数的两个或多个版本时，需要重载这个函数。当然，我们有好几次已经重载过构造函数和运算符函数，如赋值运算符。当一个函数被重载时，两种不同的函数实现必须具有不同的参数集合或不同的类型集合，使得编译器能用客户程序传递的自变量来判断应使用函数的哪个版本。

我们的 strcat 的重载版本是一个具有下面原型的函数：**void** strcat(String &add_to, **const** String &add_on)。客户程序能调用 strcat(s, t)连接字符串 s 和 t，如果参数 s 是一个 String，参数 t 可以是 C-串或 String。被重载的函数 strcat 如下实现：

```
void strcat(String &add_to, const String &add_on)
/* Post: The function concatenates String add_on onto the end of String add_to.*/
{
  const char*cfirst=add_to.c_str();
  const char*csecond=add_on.c_str();
  char*copy=new char[strlen(cfirst)+strlen(csecond)+1];
  strcpy(copy, cfirst);
  strcat(copy, csecond);
  add_to=copy;
  delete []copy;
}
```

* * * 友元函数

观察在此实现中调用的函数 strcat，它使用类型 char * 和 const char * 的自变量。C++

编译器将它识别成对 C-串函数 strcat 的调用，因为它们在自变量类型上准确地匹配。因此，我们重载的函数包含到相应库函数的一个调用，而不是到其自身的递归调用。语句 add_to=copy要求从 C-串 copy 到 String 的转换，然后应用我们重载的 String 赋值运算符。换言之，它造成两个完全的字符串复制操作。为了避免这些操作的成本，可以考虑记录这些语句。例如一种简单的解决方案是将函数 strcat 作为类 String 的友元(friend)，那么随后就可以简单地将 copy 的地址复制到 add_to. entries。

我们需要一个函数读取 String 对象，一种实现方法是重载流输入运算符<<以接受 String 参数，这种方案保持了与 C-串操作的类似性。但是我们将采用另一个可选方法，这个方法创建称为 read_in 的 String 库函数。

我们的字符串读取函数使用临时的字符 List 来收集指定为参数的流的输入。此函数然后调用合适的构造函数将这个 List 转换成 String，它假设输入由一个新行或者文件结束符终止。

```
String read_in(istream &input)
/* Post: Return a String read (as characters terminated by a newline or an end- of- file
        character) from an istream parameter. */
{
  List<char>temp;
  int size=0;
  char c;
  while((c=input.peek())!=EOF && (c=input.get())!='\n')
   temp.insert(size++, c);
  String answer(temp);
  return answer;
}
```

另一版本的函数 read_in 也有用，它使用一个辅助的引用参数来记录输入终止符，这个被重载的函数的说明遵循：

```
String read_in(istream &input, int &terminator);
```

后置条件：返回从 istream 参数读取(字符由新行或文件结束符终止)的 String，终止字符记为输出参数 terminator。

我们也发现将下面的 String 输出函数作为运算符<<的替代者也很有用。

```
void write(String &s)
/* Post: The String parameter s is written to cout.*/
{
  cout<<s.c_str()<<endl;
}
```

在下一节和后继章节，我们将使用下列其他的 String 库函数，它们的实现留作练习。

* * * 说明

```
void strcpy(String &copy, const String &original);
```

后置条件：此函数将 String original 复制到 String copy。

```
void strncpy(String &copy, const String &original, int n);
后置条件:此函数将 String original 中最多 n 个字符复制到 String copy。
```

这两个被重载的字符串处理函数被设计成具有与原来的 C-串函数相匹配的行为,但是相应的两个 C-串库函数都返回一个值,这个返回值具有类型 char * 并被设置成指向首个字符串参数,我们省略 String 库中类似函数的任何返回值。

我们需要类似的最后一个 C-串函数是 strstr,这个函数返回指向目标 C-串在文本 C-串中的首次出现位置的指针,返回的指针通常用于计算自文本起始处的偏移量。我们重载的 strstr 版本使用两个 String 参数并返回一个整数指示 target 参数在 text 参数中的首次出现的下标。

```
int strstr(const String &text, const String &target);
后置条件:若 String target 是 String text 的子串,则函数返回存储于 target 中的
          字符串在存储于 text 中的字符串中首次出现位置的数组下标。否
          则:此函数返回代码 -1。
```

练习 6.3

E1. 编写剩余的 String 方法的实现。

(1)构造函数 String()

(2)析构函数 ~String()

(3)拷贝构造函数 String(const String &copy)

(4)重载的赋值运算符。

E2. 编写下列 String 比较运算符的实现:

> < >= <= !=

E3. 编写剩余的 String 处理函数的实现。

(1)void strcpy(String &copy, const String &original);

(2)void strncpy(String &copy, const String &original, int n);

(3)int strstr(const String &text, const String &target);

* * * 回文

E4. 回文(palindrome)是一个顺读和倒读都一样的字符串,即字符串中的首个字符等于最后一个字符,第二个字符等于倒数第二个字符,等等。回文的例子包括'radar'和'ABLE WAS I ERE I SAW ELBA'。编写一个 C-+函数,测试作为引用参数传递的 String 对象是否代表一个回文。

程序设计项目 6.3

P1. 准备一个文件,它包含了 String 处理的方法和函数的实现,此文件应适合于包含到使用

字符串的任何应用程序中。

* * * 文本分析

P2. 不同的作者倾向于使用不同的词汇、不同长度的语句和不同长度的段落。此项目的意图是分析一个文本文件中的这些属性。

(1)编写一个程序读取文件，并且计算出现的各种长度的单词的个数和词的总数，然后打印出单词的平均长度和出现的各种长度的单词的百分比。对这个项目，假设单词全部由(大写和小写)字母组成并以出现的首个非字母字符结束。

(2)修改此程序，使它同时统计句子并打印出句子的总数及每句的平均单词数。假设句子以字符句点(.)、问号(?)或感叹号(!)之一的出现而结束。

(3)修改此程序，使它计算段落数并打印出段落总数和每段的平均单词数。假设段落以空行或以空格字符开始的行的出现而结束。

6.4　应用：文本编辑器

本节设计一个应用程序说明表和字符串的使用。我们的项目是开发微型文本编辑程序，这个程序只允许一些简单的命令，因而与现代文本编辑器或词处理器相比显得相当粗糙。即便如此，它仍然说明了更大更复杂的文本编辑器构造中涉及的一些基本思想。

6.4.1　说明

这个文本编辑器允许将文件读到内存中，即存储在一个缓冲区中。这个缓冲区将作为一个类的对象实现，我们称这个类为 Editor。我们认为 Editor 对象中的每行文本是一个字符串，因此 Editor 类将基于字符串类型的 List。我们将设计在缓冲区中的行上执行表操作和在单个行中的字符上执行字符串操作的编辑命令。

任何时候，由于用户可能输入插入到一行中的字符或应用于已有文本的命令，因此通常应该将文本编辑器设计成尽可能地容忍无效输入，识别无效的命令，并在执行任何过激操作前询问确认信息，如在删除整个缓冲区之前。

我们应该向编辑器实现的主程序提供所谓的命令行参数，这些参数使用标准的调用：edit infile outfile 运行编译后的程序 edit。下面给出了文本编辑器中包含的命令列表，每个命令都以所示的字母响应编辑器的提示符“??”。这些命令可用大写或小写字母键入。

* * * 命令

'R' 读取文本文件到缓冲区中，文件名在命令行中给出。缓冲区中以前的任何内容将丢失，结果，当前行将是文件的第一行。

'W' 将缓冲区的内容写入文本文件，文件名在命令行中给出，当前行或缓冲区均不改变。

'I' 插入单个新行，用户必须在恰当的提示符的响应中键入新行并提供其行号。

'D' 删除当前行并移到下一行。

'F' 从当前行开始，找到包含有用户请求的目标串的第一行。

'L' 显示当前行中的字符长度和缓冲区中的行的长度。

'C' 将用户请求的字符串修改成用户请求的替换文本，仅在当前行中有效。

'Q' 退出编辑器,立即结束。
'H' 打印解释所有命令的帮助消息,程序也接受'?'作为'H'的替代者。
'N' 下一行:在缓冲区中进一行。
'P' 上一行:在缓冲区中退一行。
'B' 开始:到缓冲区的第一行。
'E' 结束:到缓冲区的最后一行。
'G' 到缓冲区中用户指定的行号。
'S' 以用户键入的行替换当前行,函数应打印出此行以供校验并再请求新行。
'V' 查看缓冲区的全部内容,打印到终端上。

6.4.2 实现

1. 主程序

主程序的第一项任务是使用命令行参数打开输入和输出文件,如果文件能打开,主程序然后声明一个称为 buffer 的 Editor 对象并重复运行 buffer 的 Editor 方法,从用户处得到命令并处理这些命令。下面是得到的程序:

```
int main(int argc, char*argv[]) //  count, values of command-line arguments
/* Pre:  Names of input and output files are given as command-line arguments.
   Post: Reads an input file that contains lines (character strings), performs simple
         editing operations on the lines, and writes the edited version to the output
         file.
   Uses: methods of class Editor*/
{
   if (argc!=3){
      cout<<"Usage:\n\t edit inputfile outputfile"<<endl;
      exit (1);
   }
   ifstream file_in(argv[1]);  //  Declare and open the input stream.
   if (file_in==0){
      cout<<"Can't open input file"<<argv[1]<<endl;
      exit (1);
   }
   ofstream file_out(argv[2]);  //  Declare and open the output stream.
   if (file_out==0){
      cout<<"Can't open output file"<<argv[2]<<endl;
      exit (1);
   }
   Editor buffer( &file_in, &file_out);
   while(buffer.get_command())
    buffer.run_command();
}
```

2. Editor 类说明

Editor 类必须包含 String 对象的 List,并且应允许执行有效的操作以在 List 的两个方向上移动。由于事先并不能够知道缓冲区应该多大,因此为满足这些需求,Editor 类应从类

List<String>的双向链表实现中派生得到。这个派生类需要我们从主程序中调用额外的方法 get_command 和 run_command，它也需要私有数据成员来存储用户命令和到输入输出流的链接。

```
class Editor: public List<String>{
public:
  Editor(ifstream*file_in, ofstream*file_out);
  bool get_command();
  void run_command();
private:
  ifstream*infile;
  ofstream*outfile;
  char user_command;
//  auxiliary functions
  Error_code next_line();
  Error_code previous_line();
  Error_code goto_line();
  Error_code insert_line();
  Error_code substitute_line();
  Error_code change_line();
  void read_file();
  void write_file();
  void find_string();
};
```

这个类说明建立了许多辅助成员函数，这些函数将用于实现编辑器的各种命令。

构造函数将输入和输出流链接到编辑器。

```
Editor::Editor(ifstream*file_in, ofstream*file_out)
/* Post: Initialize the Editor members infile and outfile with the parameters.*/
{
  infile=file_in;
  outfile=file_out;
}
```

3. 接收命令

现在开始介绍从用户那里请求命令的方法。由于文本编辑器必须容忍无效输入，因此必须仔细地检查用户键入的命令并确信它们是有效的。由于不能期望用户在输入大小写字母上达成一致，因此第一步就是将大写字母转换成小写字母，就像来自库<cctype>中的标准例程 tolower 所做的一样。方法 get_command 需要打印出当前行，打印出提示符，从用户那里获取应答，将字母转换成小写，并检查输入是有效的。

```
bool Editor::get_command()
/* Post: Sets member user_command; returns true unless the user's command is q.
  Uses: C library function tolower.*/
{
  if (current !=NULL)
    cout <<current_position<<":"
        <<current->entry.c_str()<<" \ n??"<<flush;
  else
    cout<<"File is empty. \n??"<<flush;
```

```
  cin>>user_command;     //  ignores white space and gets command
  user_command=tolower(user_command);
  while(cin.get()!='\n')
   ;                     //  ignore user's enter key
  if (user_command=='q')
    return false;
  else
    return true;
}
```

4. 执行命令

方法 run_command 基本上是由一个大的 switch 语句组成，switch 为每个命令将工作发送给不同的函数，其中的一些函数(如 remove)仅是 List 类的成员，另一些函数则紧密依赖于相应的表处理函数，并且具有处理用户选择和错误情况的附加操作。编写查找和修改字符串的函数需要在程序设计方面做更多的努力。

```
void Editor::run_command()
/* Post: The command in user_command has been performed.
   Uses: methods and auxiliary functions of the class Editor, the class String, and the
         String processing functions.*/
{
  String temp_string;
  switch(user_command){
  case'b':
    if (empty())
      cout<<" Warning: empty buffer "<<endl;
    else
      while(previous_line()==success)
       ;
    break;
  case'c':
    if (empty())
      cout<<" Warning: Empty file"<<endl;
    else if(change_line()!=success)
      cout<<"Error: Substitution failed"<<endl;
    break;
  case'd':
    if (remove(current_position, temp_string)!=success)
      cout<<" Error: Deletion failed "<<endl;
    break;
  case'e':
    if (empty())
      cout<<" Warning: empty buffer "<<endl;
    else
      while(next_line()==success)
       ;
    break;
  case'f':
    if (empty())
      cout<<"Warning: Empty file"<<endl;
    else
```

```
      find_string();
    break;
  case'g':
    if (goto_line()!=success)
      cout<<" Warning: No such line"<<endl;
    break;
  case'? ':
  case'h':
    cout <<"Valid commands are: b(egin) c(hange) d(el) e(nd) "<<endl
         <<"f(ind) g(o) h(elp) i(nsert) l(ength) n(ext) p(rior) "<<endl
         <<"q(uit) r(ead) s(ubstitute) v(icw) w(rite) "<<endl;
  case'i':
    if (insert_line()!=success)
      cout<<"Error: Insertion failed"<<endl;
    break;
  case'l':
    cout<<"There are"<<size()<<"lines in the file."<<endl;
    if (!empty())
      cout <<"Current line length is"
           <<strlen((current->entry).c_str())<<endl;
    break;
  case'n':
    if (next_line()!=success)
      cout<<" Warning: at end of buffer"<<endl;
    break;
  case'p':
    if (previous_line()!=success)
      cout<<" Warning: at start of buffer"<<endl;
    break;
  case'r':
    read_file();
    break;
  case's':
    if (substitute_line()!=success)
      cout<<" Error: Substitution failed "<<endl;
    break;
  case'v':
    traverse(write);
    break;
  case'w':
    if (empty())
      cout<<"Warning: Empty file"<<endl;
    else
      write_file();
    break;
  default:
    cout<<"Press h or ? for help or enter a valid command:";
  }
}
```

为完成这个项目，必须依次编写由 do_command 调用的每个辅助的 Editor 函数。

5. 读写文件

由于读将破坏缓冲区中任何先前的内容，因此在开始执行此操作前要请求确认，除非缓冲区是空的。

```
void Editor::read_file()
/* Pre:  Either the Editor is empty or the user authorizes the command.
   Post: The contents of *infile are read to the Editor. Any prior contents of the Editor
         are overwritten.
   Uses: String and Editor methods and functions.*/
{
   bool proceed=true;
   if (!empty()){
      cout<<"Buffer is not empty; the read will destroy it."<<endl;
      cout<<"OK to proceed?"<<endl;
      if(proceed=user_says_yes())clear();
   }
   int line_number=0, terminal char;
   while(proceed){
     String in_string=read_in(*infile, terminal_char);
     if (terminal_char==EOF){
        proceed=false;
        if(strlen(in_string.c_str())>0)insert(line_number, in_string);
     }
     else insert(line_number++, in_string);
   }
}
```

* * * 写文件

比起 read_file，函数 write_file 显得相对简单，因此将它留作练习。

6. 插入一行

为了在当前行号处插入一个新行，首先用 6.3 节讨论的辅助的 String 函数 read_in 读取一个字符串。在读进此字符串后，用 List 的方法 insert 插入它。我们不需要直接检查缓冲区是否已满，因为这是由 List 的操作来执行的。

```
Error_code Editor::insert_line()
/* Post: A string entered by the user is inserted as a user-selected line number.
   Uses: String and Editor methods and functions.*/
{
   int line_number;
   cout<<"Insert what line number?"<<flush;
   cin >>line_number;
   while(cin.get()!='\n');
   cout<<"What is the new line to insert?"<<flush;
   String to_insert=read_in(cin);
   return insert(line_number, to_insert);
}
```

7. 查找字符串

现在开始一项更加困难的任务，即查找包含有用户指定的目标字符串的行。我们使用

String 函数 strstr 检查当前行是否包含目标，若目标没有出现在当前行中，那么查找缓冲区中的剩余部分。一旦发现了目标，则通过打印所找到的行来突出它，此时它成为当前行，并同时用一系列的上箭头(∧)指示目标在行中出现的位置。

```
void Editor::find_string()
/* Pre:  The Editor is not empty.
   Post: The current line is advanced until either it contains a copy of a user-selected
         string or it reaches the end of the Editor. If the selected string is found, the
         corresponding line is printed with the string highlighted.
   Uses: String andEditor methods and functions. */
{
   int index;
   cout<<"Enter string to search for:"<<endl;
   String search_string=read_in(cin);
   while((index=strstr(current->entry, search_string))==-1)
    if(next_line()!=success)break;
   if(index==-1)cout<<"String was not found.";
   else {
     cout<<(current->entry).c_str()<<endl;
     for(int i=0; i<index; i++)
       cout<<" ";
     for(int j=0; j<strlen(search_string.c_str()); j++)
       cout<<"^";
   }
   cout<<endl;
}
```

8. 修改字符串

与一些文本编辑器的实践相一致，我们允许查找命令建立全局的查找，从当前位置开始并持续到缓冲区的结尾。然而我们应区别对待 change_string 命令，使它仅在当前行中进行修改。用户很容易在键入目标或替换文本时出错，而 find_string 命令不做任何修改，因此这样的错误不会太严重。但如果 change_string 命令是全局有效的话，一处拼写错误就可能导致当前行以前位置的缓冲区中明显不同的部分的修改。

函数 change_line 首先从用户那里获取目标串，然后在当前行中查找它。如果未发现目标，则通知用户，否则请求用户给出替换文本，之后执行一系列的 String 和 C-串操作，将目标串从当前行删除并用替换文本去替换它。

```
Error_code Editor::change_line()
/* Pre:  The Editor is not empty.
   Post: If a user-specified string appears in the current line, it is replaced by a new
         user-selected string. Else: an Error_code is returned.
   Uses: String and Editor methods and functions.*/
{
   Error_code result=success;
   cout<<"What text segment do you want to replace?"<<flush;
   String old_text=read_in(cin);
   cout<<"What new text segment do you want to add in?"<<flush;
   String new_text=read_in(cin);
   int index=strstr(current->entry, old_text);
```

```
  if(index==-1)result=fail;
  else {
    String new_line;
    strncpy(new_line, current->entry, index);
    strcat(new_line, new_text);
    const char*old_line=(current->entry).c_str();
    strcat(new_line, (String)(old_line+index+strlen(old_text.c_str())));
    current->entry=new_line;
  }
  return result;
}
```

复杂的语句 strcat(new_line, (String)(old_line + index + strlen(old_text, c_str ()))),计算一个临时指针,它指向紧跟在被替换字符串后的 C-串 old_line,它引用的 C-串被转换成一个 String,并被立即连接到 new_line。

程序设计项目 6.4

P1. 提供下列函数,测试并运用这个文本编辑器。

(1)next_line (4)substitute_line

(2)previous_line (5)write_file

(3)goto_line

P2. 向文本编辑器加入一项功能特征,将文本如下放入到两列中。用户将选择行号范围,然后缓冲区中相应的行将被放入两个队列中,其中行的前半部分放在一个队列中,后半部分放在另一个队列中。然后将这些行从队列中移出,每次从每个队列移出一个,在它们之间用预先确定数目的空格进行结合以形成最后结果的行。(两列间的空白区称为隔条。)

6.5 数组链表

* * * 古老的语言

几门较老但广泛使用的计算机语言如 Fortran, Cobol 和 Basic 不提供动态内存分配或指针机制。然而,即使用这些语言实现,对许多问题,链表方法还是比顺序表方法更可取,例如指针的易用性而不是复制大型元素证明了它的优点。我们甚至会发现在 C++应用中,有时候最好还是使用基于数组实现的链表。本节说明如何只用整数变量和数组实现链表。

1. 方法

我们的思想是从一个大型工作区数组(或者当程序设计语言不支持结构时,用于保存每个表元素的不同部分的几个数组)开始,并将数组看成未用空间的分配。然后我们创建自己的函数来记录数组的哪些部分是未用的,并以期望的顺序将数组中的元素链接起来。

* * * 动态内存

数组链表的这种实现方法总会失去的一个特征是动态存储分配,因为我们必须事先决定分配给每个数组多大的空间。链表的其余特征如重新排列大型元素的灵活性或易于在表

中任何位置插入或删除等仍然可用，因此链表仍然证明是一种重要的方法。

数组链表甚至在像 C++这样提供了指针和动态内存分配的语言中也证明是重要的。证明数组更可取的应用包括：

* * * 优点

- 表中的元素个数事先已知；
- 链经常需要重排，但进行相对少的增加或删除；
- 同样的数据有时最适于作为链表对待，而其他的时候适于作为顺序表对待。

* * * 多重链接

这样的一个应用例子如图 6.6 所示，它显示了学生记录系统的一小部分。鉴别号码按“先来先服务”的原则分配给学生，因此任何特定课程中的名字或成绩都不是按照特别的顺序。给定一个鉴别号，可用这个鉴别号作为索引在数组中立即找到学生记录。然而有时希望按学生姓名的字母顺序打印出学生记录，这个操作可以通过循着存储在数组 next_name 中的链来实现。类似地，可以循着恰当的数组中的链对学生记录按任何课程中的成绩进行排序。

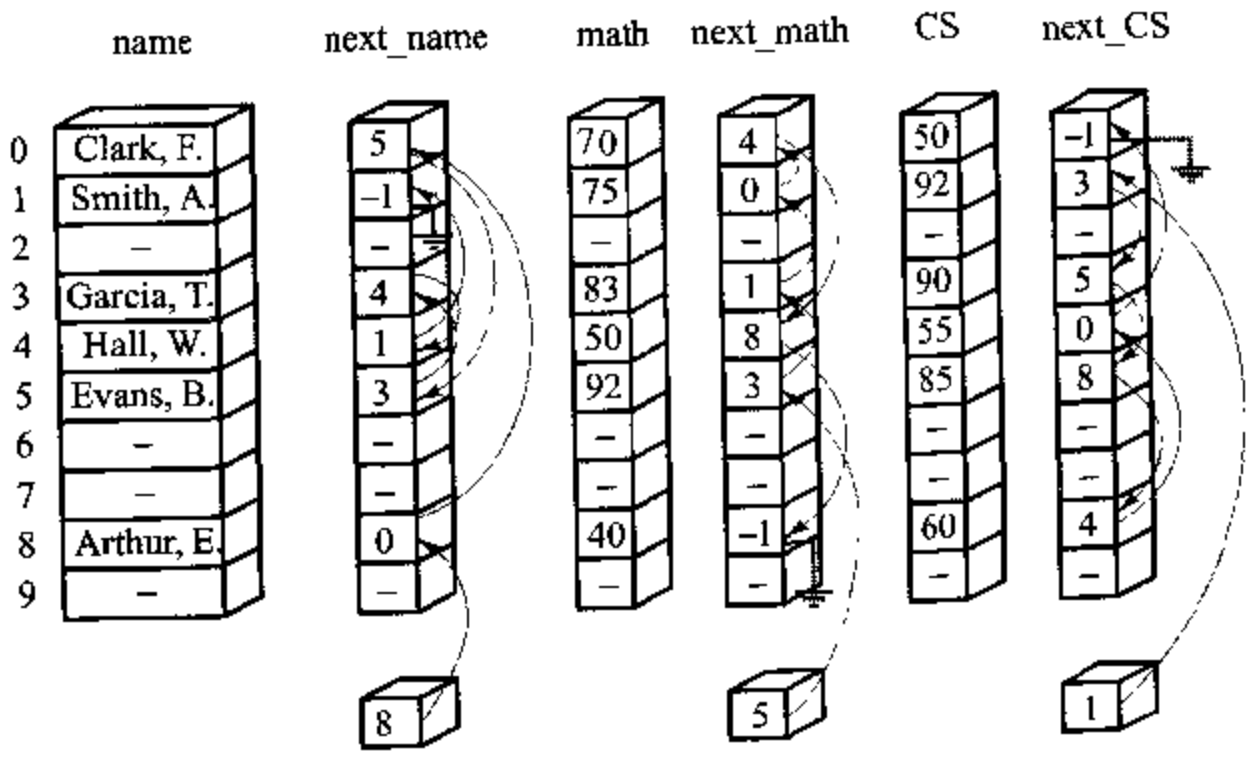


图 6.6　数组链表

为证明链表的这种实现如何行得通，让我们遍历图 6.6 的第一部分中所显示的链表 next_name，其表头（在表格的下面显示）包含数值 8，这意味着此元素在位置 8 中，Arthur，E. 是表的第一个元素。Next_name 的位置 8 包含数值 0，它意味着位置 0 中的名字 Clark，F. 是下一个。在位置 0，next_name 包含的值为 5，因此 Evans，B. 是接下来的元素。位置 5 指向位置 3（Garcia，T.），位置 3 指向位置 4（Hall，W.），位置 4 指向位置 1（Smith，A.）。在位置 1 中，next_name 的值为 -1，意味着位置 1 是链表的最后一个元素。

类似地，数组 next_math 描述的链表在数组 math 中以降序给出了分数，第一个元素是 5，它指向元素 3，按照链表的顺序接下来的结点是 1，0，4 和 8。

同样，由 next_CS 描述的链表中出现的结点的顺序是 1，3，5，8，4 和 0。

* * * 共享表和随机访问

如图 6.6 所示，数组链表能够获得修改链表的灵活性和在几个链表间共享同一信息域

(如图 6.6 中的姓名)的能力,并能通过使用下标直接访问元素而具有随机访问的优点,否则这一优点只是对顺序表可用。

* * * 下标

在数组链表的实现中,指针成为相对于数组开始处的下标(index),表的链存储在一个数组中,表的每个元素给出了数组中存储的表的下一个元素的下标。为将这些下标与动态存储中链表的指针区分开来,我们将数组中的链称为下标,而将指针一词保留给动态存储中的指针使用。

为了编写程序,我们能为每个链表声明两个数组,其中 entry[]用于保存结点的信息,而 next_node[]用于给出下一个结点的下标。对大多数应用来说,entry 是结构化元素的一个数组,或者当程序设计语言不支持结构时分成几个数组。数组 entry 和 next_node 都将从 0 到 max_list-1 编排下标,这里的 max_list 是一个符号常量。

* * * 空下标

因为下标从 0 开始,我们做出另一个武断的选择,使用下标值 -1 指示表的结尾,就像动态存储中使用的指针值 NULL 一样,这一选择也在图 6.6 中给予了图解说明。

我们应花一些时间对图 6.6 进行从头到尾的跟踪,检查显示的下标值与浅灰色箭头之间的对应关系,这些箭头从每个元素指向其后继者。

2. 操作:空间管理

为获得数组链表的特色,让我们用这种实现重新编写本章的一些函数。

* * * 链表的工作区

我们的首要任务是创建一个可用工作区的表,编写辅助函数获得新的结点并返回一个结点给可用的缓冲区。为了保持与 6.2 节程序设计的一致性,我们将稍微改变观点,要使用的所有空间将来自一个称为 workspace 的数组,并将 workspace 的元素看成结点,然后设计一种称为 Node 的数据类型来存储元素数据。每个 Node 是具有两个成员的一种结构,这两个成员分别是 List_entry 类型的 entry 和 index 类型的 next。类型 index 作为整数实现,但它的值被解释成数组的位置,因此代替了其他链表中的指针类型。

workspace 中的可用空间有两种:

- 首先,有未被分配的结点;
- 其次,有以前使用过但现在已释放的结点。

* * * 已用位置的计算

我们最初在数组的起始处分配空间,因此能够用一个下标 last_used 记录某个时刻已有多少空间被使用,last_used 指示某个时刻上个结点已用掉的位置,比 last_used 大的下标值所指的位置都未被分配。

* * * 以前用过的空间的链栈

对已经使用过并返回给可用空间的结点,需要采用某种链式结构从一个结点访问下一个结点。由于链栈是这种结构最简单的形式,因此采用一个链栈记录以前使用过但返回给可用空间的结点。这个栈将用数组 workspace 的结点中的 next 下标进行链接。

为了跟踪可用空间的栈,需要一个整数变量 available 给出栈顶的下标。如果这个栈是空的(它将表示成 available==-1),那么就需要获取一个新的 Node,即数组中迄今未分配给

任何 Node 使用过的位置，可以通过增加下标变量 last_used 来获取这个新的 Node，last_used 用于统计数组中已用于保存表元素的位置总数。当 last_used 达到 max_list-1(假设的数组大小的边界)和 available==-1 时，则工作区已满且没有更多的空间可用于分配。

* * * 受保护的成员

我们将数组工作区下标 available 和 last_used 声明成 List 类的受保护的数据成员。当初始化一个 List 对象时，成员 available 和 last_used 都应当被初始化成 -1，available 指示以前用过但现在可用的空间的栈是空的，last_used 指示还未从数组中分配过空间。

可用空间表在图 6.7 中给出图示，数组 next_node 左边显示的箭头描述了一个链表，它生成表中按字母顺序排列的姓名。数组 next_node 右边的箭头与头变量 available 一起，说明了(以前用过但现在)可用的空间栈中的结点。注意可用空间表中出现的下标正是位置 10 或以前未分配给数组 workspace 中的名字的下标。最后，位置 11 或更后的位置中均未指派元素，这个事实由值 last_used=10 指出。如果我们要向 List 中插入额外的姓名，首先要从栈中用栈顶 available 弹出结点，并且仅当栈空时才增加 last_used，从而在以前未使用的空间中插入一个姓名。

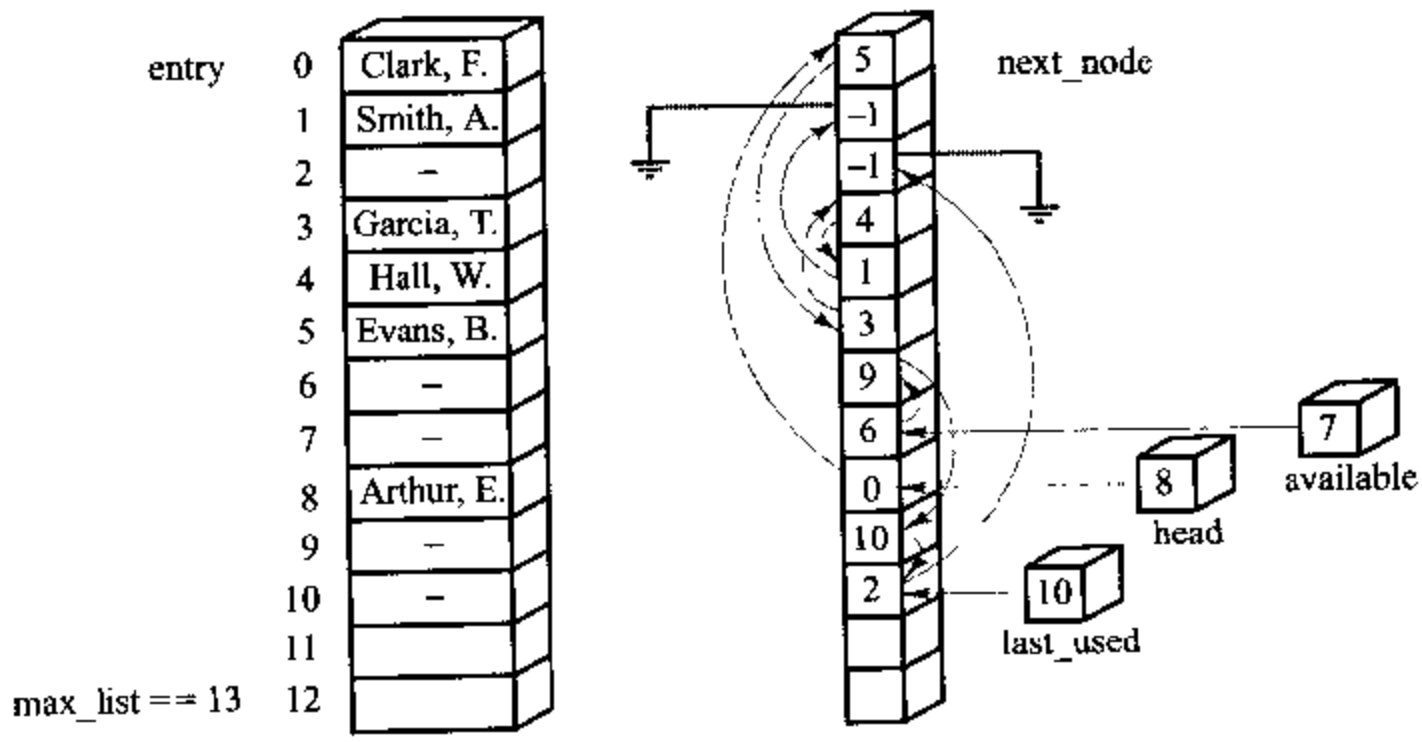


图 6.7　具有可用空间的数组和栈

我们做出的决定转换成下面的声明，它们将被放入链表的说明文件中：

```
typedef int index;
const int max_list= 7;      //  small value for testing purposes

template<class List_entry>
class Node {
public:
  List_entry entry;
  index next;
};

template<class List_entry>
class List {
public:
// Methods of the List ADT
```

```
  List();
  int size()const;
  bool full()const;
  bool empty()const;
  void clear();
  void traverse(void (*visit)(List_entry &));
  Error_code retrieve(int position, List_entry &x)const;
  Error_code replace(int position, const List_entry &x);
  Error_code remove(int position, List_entry &x);
  Error_code insert(int position, const List_entry &x);
protected:
//  Data members
  Node<List_entry>workspace[max_list];
  index available, last_used, head;
  int count;
//  Auxiliary member functions
  index new_node();
  void delete_node(index n);
  int current_position(index n)const;
  index set_position(int position)const;
};
```

* * * 兼容性

我们观察到公共可用的方法正与我们的其他表实现中的那些方法一样，这意味着新的实现与任何早先的 List ADT 实现是可以互换的。我们加入了许多受保护的成员函数，它们中的大多数用于管理 workspace 中的结点，我们将它们用作构造方法的工具，但它们对客户程序是不可访问的。用这些声明，现在能够编写跟踪未用空间的函数。函数 new_node 和 delete_node 起到 C++控制符 new 和 delete 的作用，因此，new_node 从 workspace 返回一个以前未分配过的下标。这些函数采用下面的形式：

```
template<class List_entry>
index List<List_entry>::new_node()
/* Post: The index of the first available Node in workspace is returned; the data
         members available, last_used, andworkspace are updated as necessary. If the
         workspace is already full, -1 is returned. */
{
  index new_index;
  if (available !=-1){
    new_index=available;
    available=workspace[available].next;
  } else if(last_used<max_list-1){
    new_index=++last_used;
  } else return-1;
  workspace[new_index].next=-1;
  return new_index;
}


template<class List_entry>
void List<List_entry>::delete_node(index old_index)
/* Pre:  The List has a Node stored at index old_index.
   Post: The List index old_index is pushed onto the linked stack of available space;
```

```
        available, last_used, and workspace are updated as necessary. */
{
  index previous;
  if (old_index==head) head=workspace[old_index].next;
  else {
    previous=set_position(current_position(old_index)-1);
    workspace[previous].next=workspace[old_index].next;
  }
  workspace[old_index].next=available;
  available=old_index;
}
```

当然，这两个函数只是出栈和入栈，如果我们愿意的话，可以编写用于处理栈的函数并使用那些函数。

其他受保护的成员函数是 set_position 和 current_position。就像我们在更早的实现中一样，set_position 操作用于定位 workspace 中的下标，此 workspace 存储具有给定位置号的表元素。current_position 操作使用 workspace 中的一个下标作为自己的参数，计算存储在此处的任何表元素的位置。我们将这些操作留作练习，它们的说明如下：

```
index List<list_entry> :: set_position(int position) const;
```
前置条件：position 是表中的一个有效的位置，0≤position < count。
后置条件：返回表中 position 处的结点的下标。

```
int List<List_entry> :: current_position(int n) const;
```
后置条件：返回存储于下标 n 的结点的位置号，或者当没有这样的结点时返回−1。

3. 其他操作

可以通过转换链表方法，继续完成数组链表的所有方法的编码，其中的大多数方法将留作练习。但是为了提供模型，我们编写了转换函数来遍历 List 和将一个新元素插入到 List 中。

```
template<class List_entry>
void List<List_entry> :: traverse(void (*visit)(List_entry &))
/* Post: The action specified by function *visit has been performed on every entry of the
         List, beginning at position 0 and doing each in turn. */
{
  for(index n=head; n !=-1; n=workspace[n].next)
    (*visit)(workspace[n].entry);
}
```

将这个方法与 6.2 节提供的用指针和动态内存实现的简单链表的相应方法进行比较，很快能够发现此实现中的每个语句只是以前的实现中的相应语句的简单转换。类似的转换过程可以将以前的插入方法转换成基于数组链表实现的插入方法。

```
template<class List_entry>
Error_code List<List_entry> :: insert(int position, const List_entry &x)
```

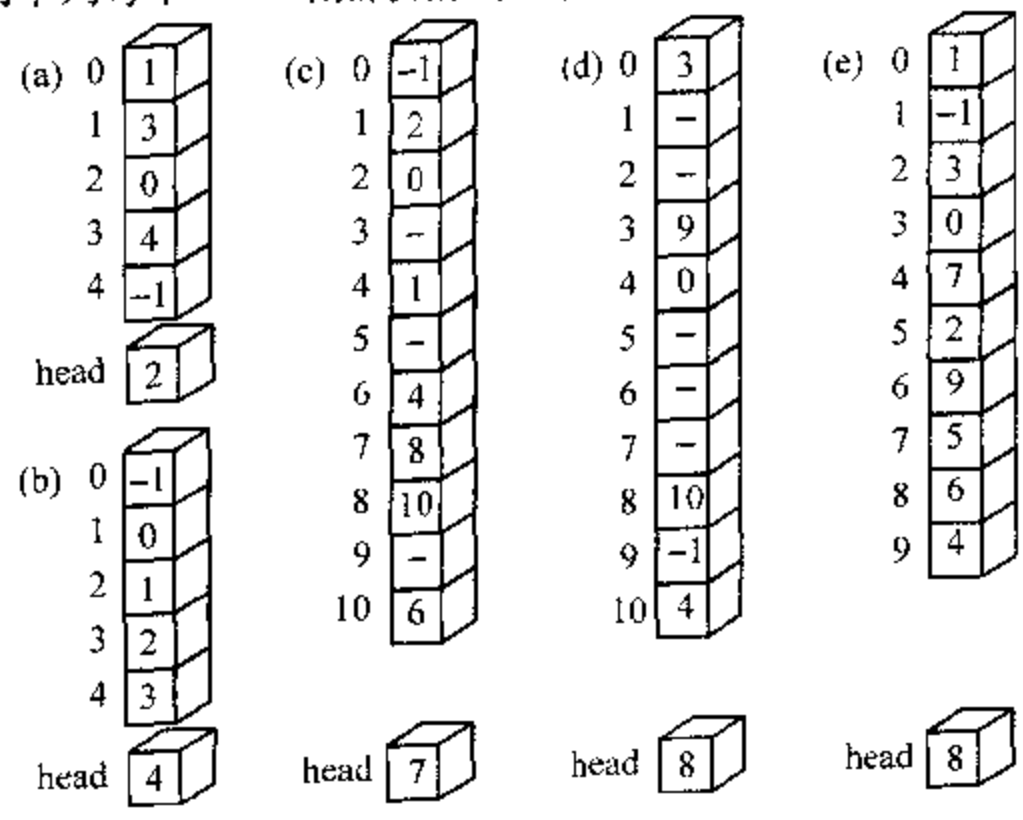
```
/* Post: If the List is not full and 0 <= position <= n, where n is the number of entries in
         the List, the function succeeds: Any entry formerly at position and all later
         entries have their position numbers increased by 1 and x is inserted at
         position of the List.
         Else: the function fails with a diagnostic error code. */
{
  index new_index, previous, following;
  if(position<0||position>count)return range_error;
  if (position>0){
    previous=set_position(position-1);
    following=workspace[previous].next;
  }
  else following=head;
  if((new_index=new_node())==-1)return overflow;
  workspace[new_index].entry=x;
  workspace[new_index].next=following;
  if (position==0)
    head=new_index;
  else
    workspace[previous].next=new_index;
  count++;
  return success;
}
```

4. 链表的变种

带下标的数组不限于实现简单链表，它们对双向链表或链表的任何其他变种都同样有效。实际上，对于双向链表，下标算术运算能力使得实现（下标既使用正数值，也使用负数值）中单一的下标域能同时包含前向和后向链。（见练习 E5）

练习 6.5

E1. 画出箭头说明下列每个 next 结点表格当中表元素是如何链在一起的。



E2. 构造 next 表格，说明下面的每个表如何链接成字母顺序，同时在每种情况下，给出表头的变量 head 的值。

(1) 1 array	(3) 1 the	(4) 1 London
2 stack	2 of	2 England
3 queue	3 and	3 Rome
4 list	4 to	4 Italy
5 deque	5 a	5 Madrid
6 scroll	6 in	6 Spain
	7 that	7 Oslo
(2) 1 push	8 is	8 Norway
2 pop	9 I	9 Paris
3 add	10 it	10 France
4 remove	11 for	11 Warsaw
5 insert	12 as	12 Poland

E3. 对前一问题(4)部分中的城市和国家列表，构造一个 next 结点表，它生成一个链表，包含以字母顺序排列的所有城市，后面紧跟以字母顺序排列的所有国家。

E4. 编写数组链表的下列函数，确信每个函数符合 6.1 节中给出的说明和本节中的声明。

(1)set_position　(5)full　(8)remove
(2)List(构造函数)　(6)size　(9)replace
(3)clear　(7)retrieve　(10)current_position
(4)empty

E5. 使用仅有一个下标 next 的工作区数组去实现双向链表是可能的，即不必在结点中保留一个单独的域 back，back 域是数组的一部分，其目的是为了找到后向链。这种思想是不再将表中的下一元素的下标放入 workspace[current]，而是用一个成员 workspace[current]. difference 给出下个元素的下标减去 current 前趋元素的下标。我们也必须为表中的后续结点维持两个指针，分别是 current 下标和链表中恰在 current 之前的结点的下标 previous。为找到表的下一个元素，我们计算 workspace[current]. difference+previous；类似地，为找到 previous 前的元素，我们计算 current - workspace[previous]. difference；这种表的一个例子显示在下图的第一部分中，每个框中显示了存储在 difference 中的值，在右边是相应的下标值的计算。

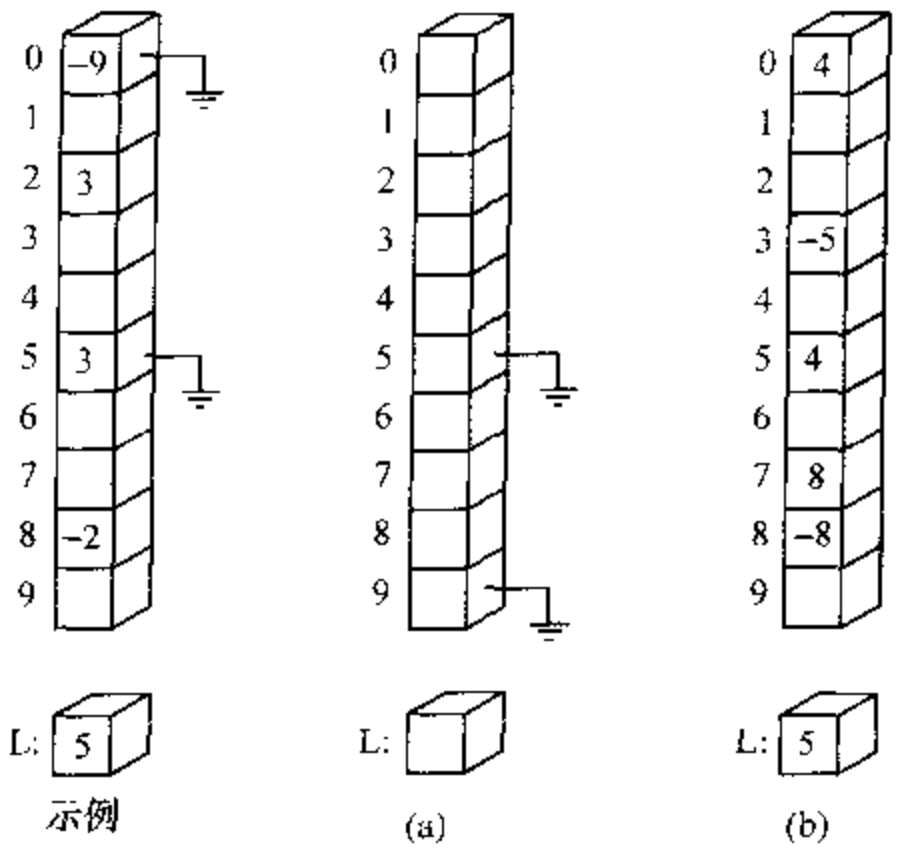
(1)对上图第二部分中显示的双向链表，说明将存储在 list. head 中和工作区中被占据结点的 difference 域中的值。

(2)对上图第三部分中显示的 list. head 和 difference 值，画出展示相应的双向链表的链。

(3)用这种实现，编写函数 set_position。

(4)用这种实现，编写函数 insert。

(5)用这种实现，编写函数 remove。



6.6 应用:生成排列

本章中最后一个示例项目以高度专用的方式说明通用表和数组链表的使用。这个项目将尽可能高效地生成 n 个对象的所有 $n!$ 种排列。n 个不同对象的排列是以不同顺序摆放这些对象的所有方法。①

n 个对象为何有 $n!$ 种排列的原因是因为我们可以选择 n 个对象中的任何一个作为第一个,然后选择 $n-1$ 个剩余对象中的任何一个作为第二个,以此类推。这些选择是独立的,因此选择的数目要相乘。如果将数 $n!$ 看作积 $n!=1\times2\times3\cdots\times n$,那么乘法的处理过程可以画作图 6.8 中的树。(暂时忽略标签。)

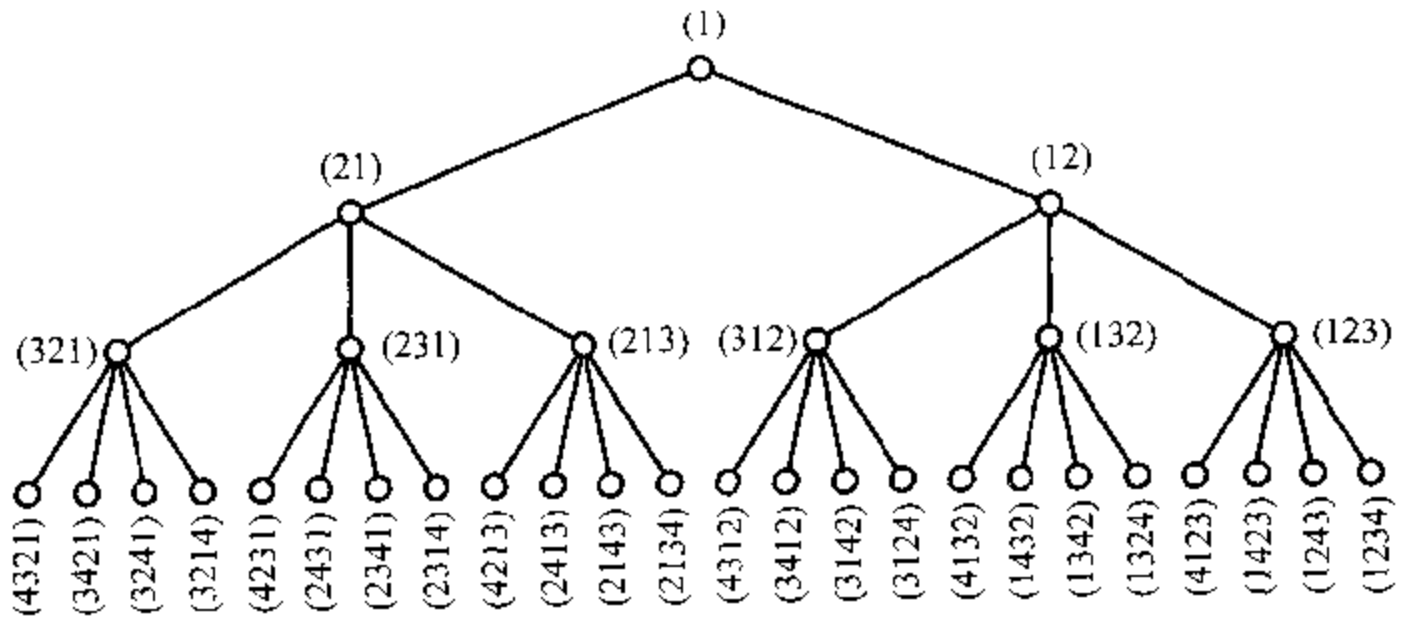


图 6.8 $n=4$ 时乘法的排列生成

① 关于排列的更多的信息,见附录 A。

1. 思想

我们可以用图 6.8 中的标签给出的结点确定排列。顶层是 1 自身，通过在 1 的左边和右边插入 2，得到{1,2}的两个排列。类似地，从排列(2,1)或(1,2)中的某一个开始，并将 3 插入到三个可能的位置之一(左、中或者右)，可以得到{1,2,3}的六个排列。生成{1,2,…,k}的排列的任务现在能够概括为：

> 取{1,2,…,$k-1$}的一个给定的排列并将它的元素放入一个表中，依次在此表的 k 个可能的位置上插入 k，因而得到{1,2,…,k}的 k 个不同的排列。

这个算法说明利用递归来完成被暂时延缓的任务，也就是说，我们应编写一个函数，它首先将 1 插入到一个空表中，然后使用递归的调用将从 2 到 n 的剩余的数字插入到表中。第一个递归将 2 插入到仅包含 1 的表中，并在递归调用中进一步插入其他数字。最后在第 n 次递归调用时，将插入整数 n。这样，从作为动机的树结构开始，现在已能设计出将给定的树变成递归树的算法。

2. 细化

现在以稍微更正式的术语重新描述这个算法。我们的函数调用为 permute(1, n)，它表示插入 1～n 的所有整数以建立所有 $n!$ 个排列。当插入整数 k 时，其余的任务是：

```
void permute(int k, int n)
/* Pre:  1 through k-1 are already in the permutation list;
   Post: inserts the integers from k through n into the permutation list*/
{
   for              //  each of the k possible positions in the list
   {
      //  Insert k into the given position.
      if(k==n)process_permutation;
      else permute(k+1, n);
      //  Remove k from the given position.
   }
}
```

函数 process_permutation 可以对{1,2,…,n}的全部排列按任何要求进行处理，我们可能只希望将它打印出来，或者可能希望将它作为某个其他任务的输入。

3. 一般过程

为了将此算法转换成 C++，需要修改某些符号。我们将在类型为 List<**int**>的变量 permutation 中创建排列，将采用 new_entry 而不是 k 来表示被插入的整数，并且用 degree 而不是 n 来表示要排列的对象总数。于是得到下面的函数：

```
void permute(int new_entry, int degree, List<int>&permutation)
/* Pre:  permutation contains a permutation with entries in positionsl through new_
         entry-1.
   Post: All permutations with degree entries, built from the given permutation, have
         been constructed and processed.
```

```
  Uses: permute recursively, process permutation, and list functions.*/
{
  for (int current=0; current<permutation.size()+1; current++){
    permutation.insert(current, new_entry);
    if (new_entry==degree)
      process_permutation(permutation);
    else
      permute(new_entry+1, degree, permutation);
    permutation.remove(current, new_entry);
  }
}
```

将此函数嵌入到能够运行的程序中的任务将留作一个项目，对所需的表函数，6.2 节的任何实现均可接受。

4. 数据结构：优化

$n!$ 随 n 增长得非常快，排列数确实随 n 上升得非常快，因此这个项目是值得优化以加快速度的少数几个应用之一，特别是如果希望使用这个程序去研究关于生成排列的有趣的问题时更是如此。

*** 数组链表

我们尽可能地增加程序速度而对数据的表达做一些改变，即使是损害一些可读性。可以用一个表存放待排列的数，作为 permutation 数组，这个表对函数的递归调用是可用的，并且每个递归调用都更新这个表中的元素。由于必须不断地往表中插入或从表中删除元素，因此比起将元素保存在顺序表中，链式存储更加灵活。但表中的元素总数决不会超过 n，因此可以将链表存放在数组中而不是使用动态内存分配以(可能)提高效率，我们的链因此是与数组起始位置相关的下标。此外，采用数组，每个元素的下标就像它被指派的那样，将碰巧等于被插入的数的值，因此就不再有任何必要保存这个数字值，所以在数组中只需保存链。

在图 6.9 中给出了这种将排列表示为数组链表的图解说明，最上面的图将排列(3,2,1,4)表示成一个链表；第二个图将它表示成数组链表；第三个图省略了被排列的实际元素，因为它在这个数组中与位置是相同的，因此仅保存描述此链表的链。

*** 人为结点

如果我们在表的起始处配置一个人为的首结点(artificial first node)，则可以进一步简化插入和删除，因此在(实际的)表的起始处的插入和删除可以按照与其他位置的情况同样对待，即总是在一个结点后进行插入或删除。所以通过使用所有这些特别条件并在函数 permute 中编写插入和删除，而不是使用泛化的表实现，我们能够获得提高的效率。

5. 最后的程序

根据这些决定，能够编写出优化版本的 permute。

```
void permute(int new_entry, int degree, int*permutation)
/* Pre:  permutation contains a linked permutation with entries in positions 1 through
         new_entry-1.
 Post: All permutations with degree entries, built from the given permutation, have
         been constructed and processed.
```

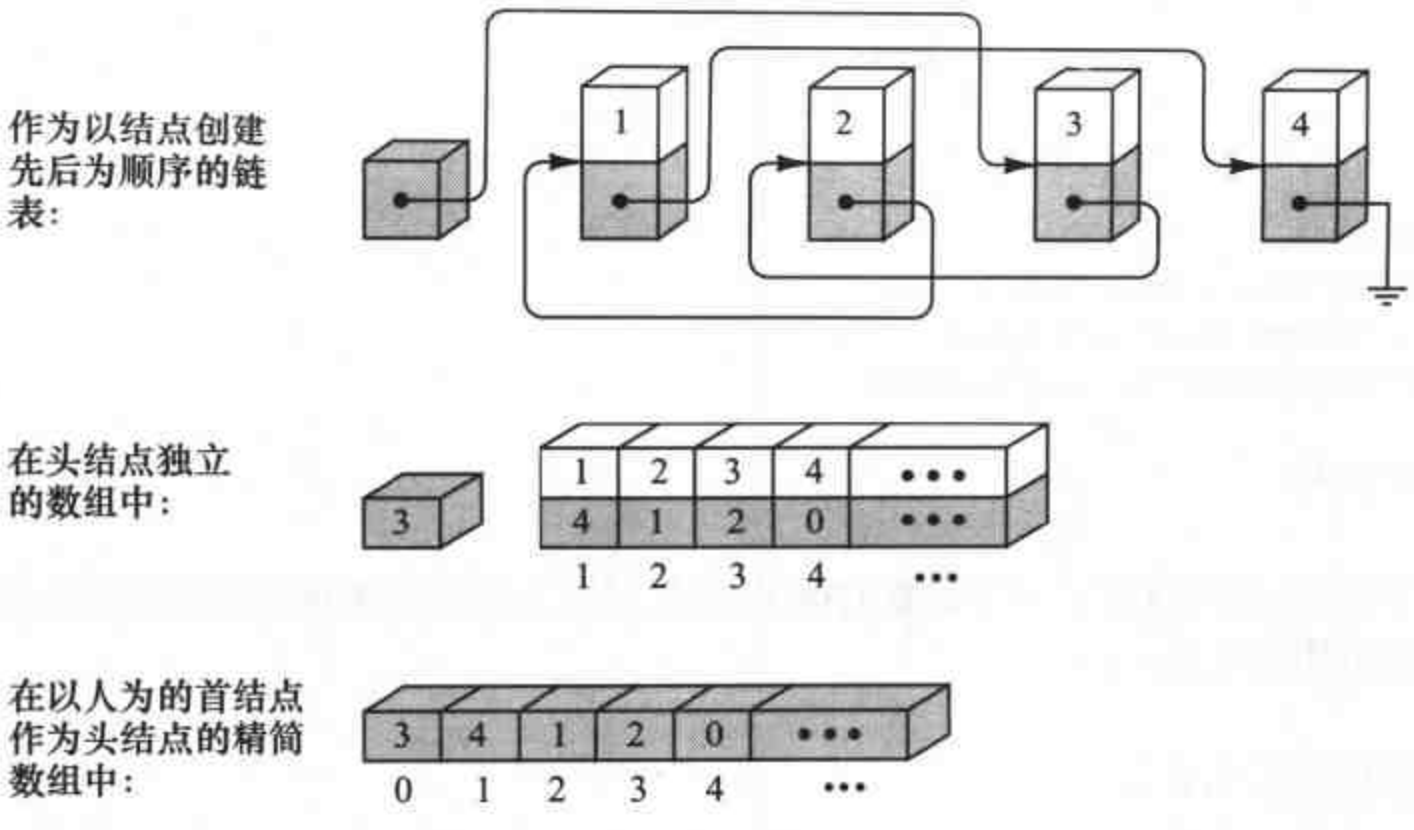


图 6.9　排列作为数组链表

```
   Uses: Functions permute(recursively) and process_permutation.*/
{
   int current=0;
   do {
      permutation[new_entry]=permutation[current];
      permutation[current]=new_entry;
      if (new_entry==degree)
         process_permutation(permutation);
      else
         permute(new_entry+1, degree, permutation);
      permutation[current]=permutation[new_entry];
      current=permutation[current];
   } while(current!=0);
}
```

主程序除了建立声明和初始化此过程以外,其他的工作很少。

```
main()
/* Pre:  The user specifies the degree of permutations to construct.
   Post: All permutations of a user-supplied degree are printed to the terminal.*/
{
   int degree;
   int permutation[max_degree+1];
   cout<<"Number of elements to permute?";
   cin  >>degree;
   if (degree<1 || degree>max_degree)
      cout<<"Number must be between 1 and"<<max_degree<<endl;
   else {
      permutation[0]=0;
      permute(1, degree, permutation);
   }
}
```

数组 permutation 描述了一个指针链表，并且没有包含待排列的对象，例如，如果希望打印出待排列的整数 1，…，n，则辅助函数变成：

```
void process_permutation(int*permutation)
/* Pre:  permutation is in linked form.
   Post: The permutation has been printed to the terminal. */
{
   int current=0;
   while(permutation[current]!=0){
    cout<<permutation[current]<<"";
    current=permutation[current];
   }
   cout<<endl;
}
```

有了这个函数，我们就有了一个完整的程序，并且实际上这个程序是以最快速度生成排列的最有效的可用程序之一。

程序设计项目 6.6

P1. 完成一种版本的排列生成程序，它使用某一种通用的表实现，写出它的主程序和在终端上打印排列的函数 process_permutation。在测试完程序后，隐藏打印排列的函数，并包含编译器提供的 CPU 计时函数。比较你的程序与 6.2 节中每个表实现之间的性能，同时将它的性能与课本中编写的优化后的程序的性能进行比较。

P2. 修改通用版本的 permute，使得从任意排列到生成的下--个排列，每个数所占用的位置靠左或靠右的变化不超过一个位置。（这是鸣钟术（教堂钟声的变化）的一条规则的简化的形式。）

启示和易犯的错误

1. 使用 C++模板实现泛型数据结构。
2. 不要混淆顺序表和数组。
3. 在设计算法时选择数据结构，并避免做出不成熟的决定。
4. 总是小心极端情况并优雅地处理它们。跟踪整个程序以判断数据结构为空或已满时会发生什么。
5. 在代码完善地运行前不要优化它，仅在确实需要提高效率的情况下优化代码。首先尝试数据结构的简单的实现，仅在简单的数据结构实现证明效率低下时才换用更复杂的实现。
6. 当采用通用表时，首先明确地决定需要什么操作，然后选择能最简单地完成那些操作的实现。
7. 在表的链式和顺序实现之间做出选择时，考虑表上必要的操作。链表对插入、删除和重排更加灵活，而顺序表允许随机访问。
8. 当表中的项较小且算法简单时，顺序表一般需要较少的计算机内存、计算机时间和程序设计工作。当表中存放着较大的数据元素时，链表--般更节省空间和时间，并经常也节

省程序设计工作。

9. 动态内存和指针允许程序动态地适应广泛的应用规模，且提供了不同数据结构间内存分配的灵活性。静态内存(数组和下标)有时对能事先完全确定大小的应用更有效。
10. 关于动态内存中用链表进行程序设计的建议，见第 4 章中的指导原则。
11. 避免为了复杂而复杂，如果简单的方法已足够用，则使用简单的方法。
12. 避免做重复的工作，如果现成的类模板或函数已足够用，则使用现成的函数。

复习题

6.1

1. 能用于通用表的哪些操作也能用于队列？用于栈？
2. 列举出三种对通用表是可能的但对于栈或队列却不允许的操作。

6.2

3. 如果表中的项是整数(每个是一个字)，若(1)表顺序保存在一个 90%满的数组中，(2)表顺序保存在一个 40%满的数组中，以及(3)表作为链表保存(这里每个指针占一个字)，比较总共所需要的空间量。
4. 当表中的项是各占用 200 个字的元素时，重复前一练习中的比较。
5. 链表同顺序表比较主要的缺点是什么？

6.3

6. C-串的主要缺点是什么？

6.5

7. 以带下标的数组而不是带指针的动态内存来实现链表的一些理由是什么？

进阶参考书目

为栈和队列列出的参考书目仍适用于本章，特别是对许多与表操作有关的主题，附加信息、历史记录和数学分析的最好资源是 Knuth 的第 1 卷，但这本书没有考虑数据抽象的原理。

关于 C++标准字符串实现和库的细节，参见下面这本书的第 20 章：B. Stroustrup, *The C++ Programming Language*, third edition, Addison-Wesley, Reading, Mass., 1997.

对字符串到 C-串转换的各种实现策略的优缺点的仔细讨论，见：Scott Meyers, *Effective* C++, second edition, Addison-Wesley, Reading, Mass., 1997.

通过向链表中插入来生成排列的算法发表于 ACM *SIGCSE Bulletin* 14 (February 1982), 92-96 中。关于生成排列的许多方法的概论是：R. Sedgewick, "Permutation generation methods", *Computing Surveys* 9 (1977), 137-164; addenda, ibid., 314-317. R. W. Topor, "Functional programs for generating permutations", *Computer Journal* 25 (1982), 257-263.

排列在鸣钟术(改变钟声)中的应用产生了一些值得计算机研究的有趣问题，更多信息的极好的来源是：F. J. Budden, *The Fascination of Groups*, Cambridge University Press, Cambridge, England, 1972, pp. 451-479.

第7章 查 找

本章介绍在表中进行查找以发现特定元素这个问题。我们的讨论集中于两个著名的算法:顺序查找和二分查找。我们将设计几个复杂的数学工具,用于论证算法的正确性并计算它们所做的工作量。这些数学工具包括不变断言、比较树、大-O、Θ和Ω符号。最后,我们将得到查找算法的下界,它表明任何查找算法所做的工作量必须至少同二分查找一样多。

7.1 查找:引言和符号

信息检索是最重要的计算机应用之一,例如:给出一个名字要求得到相关的电话清单,给出一个账号要求得到以这个账号发生的交易,给出一个雇员的名字或号码要求得到这个雇员的人事档案。

* * * 键和记录

1. 键

在上面这些例子及大量的其他例子中,都是给出一部分信息,我们称之为键(key),并且要求找出包含有与这个键相关的其他信息的记录(record)。我们应该允许同一个键有多条记录或者对给定的键没有任何相关的记录这两种可能性,见图7.1。

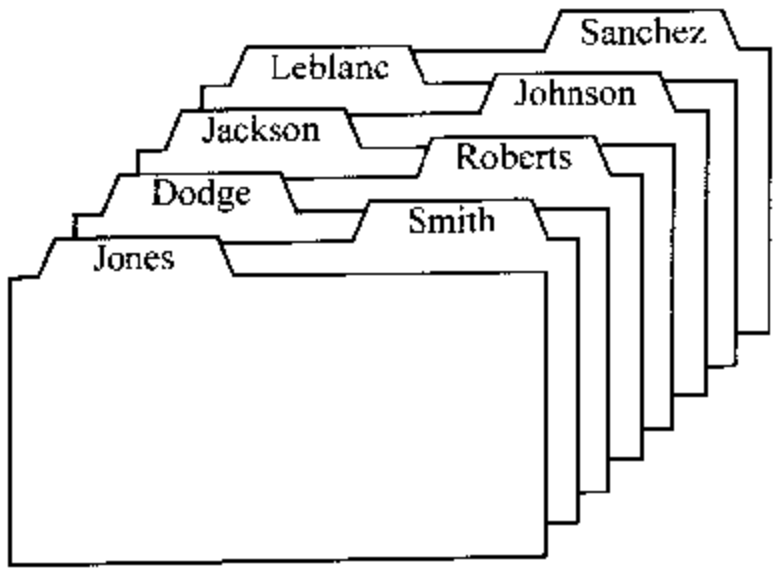


图7.1 记录和键

2. 分析

查找用于定位记录的键经常是程序中最耗时的操作,因此排列记录的方法和用于查找的方法的选择对程序的性能会有实质性的差别。因为这个原因,我们要在本章花些时间去研究我们所设计的每个算法的工作量。我们发现,一个键与其他键的比较次数提供了极好的标准,用于度量算法要做的工作量和它运行时所需的计算机时间总量。

3. 外部和内部查找

查找问题自然地分成两种情况。如果有许多记录，可能每条记录都很大，那么就有必要将这些记录存储在计算机内存以外的磁盘或磁带文件上，这种情况称为外部查找。在另一种情况中，被查找的记录完全存储在计算机内存中，这种情况称为内部查找。在本书中，我们仅考虑内部查找。虽然在本章和下一章中设计的许多方法都可用于外部查找，但是对外部查找方法的全面研究已超出了本书的范围。

4. 用C++实现

为了用C++实现我们的程序，特设立一些约定。

*** 仅顺序式

某些查找算法当应用于链表实现时效率较低，因此，在本章中默认任何表都有顺序实现。对链式结构的查找是第10章关心的主要问题，因此将推迟到那时再来考虑链式结构。

本章只关心表的查找。我们的一些程序所查找的表满足第6章的ADT说明，而其他程序则用于种类稍有不同的表。但是在每种情况中，总是查找顺序记录表，我们一般称这个表为the_list。存于表中被查找的记录必须符合下列最低标准：

- 每个记录与一个键相关；
- 可以比较键的相等或相对次序关系；
- 记录可互相比较，或者首先将记录转换成相关的键，然后根据键进行比较。

因此我们将实现的查找程序应该能用于反映这种行为的Record类型的对象。特别地，存在一种相关的称为Key的类型(它可能与Record一样)以及将Record变成其相关的Key的转换操作。在应用程序中，这个转换操作可以是下列之一：

- 类Record的方法，以 **operator** Key() **const**; 声明。
- 类Key的构造函数，以 Key(**const** Record &); 声明。
- 或者，如果类Key和Record是一样的，则不必定义转换，因为任何Record都自动地是Key。

*** 比较运算符

我们要求类型Key的任何一对对象可以用下列标准的运算符进行比较：==，!=，<，>，<=，>=。另外，由于任何Record可能被编译器转换成Key，因此Key的比较运算符适用于记录间的比较或者记录到键的比较。

例如，为了选择已有的类型作为记录或键，客户程序可以用如下的类型定义语句：

```
typedef int Key;
typedef int Record;
```

作为选择，客户程序能定义新类，它基于下列框架去体现适当的行为：

```
// Definition of a Key class:
  class Key{
    public:
    // Add any constructors and methods for key data.
    private:
    // Add declaration of key data members here.
```

```
};
//  Declare overloaded comparison operators for keys:
  bool operator==(const Key &x, const Key &y);
  bool operator> (const Key &x, const Key &y);
  bool operator< (const Key &x, const Key &y);
  bool operator>=(const Key &x, const Key &y);
  bool operator<=(const Key &x, const Key &y);
  bool operator!=(const Key &x, const Key &y);
//  Definition of a Record class:
  class Record{
    public:
     operator Key();          //  Implicit conversion from Record to Key.
    //  Add any constructors and methods for Record objects.
    private:
    //  Add data components.
};
```

特别指出，我们并没有假设 Record 必须有 Key 对象作为数据成员，虽然这一点经常是事实。我们仅假设编译器能将 Record 变成其对应的 Key。

5. 参数

* * * 参数

* * * 目标

我们所编写的每个查找函数有两个输入参数，其中第一个参数给出被查找的表，第二个参数给出待查找的键，这个键称为查找的目标(target)。

查找函数也有一个输出参数和一个返回值。返回值的类型为 Error_code，指示查找过程是否成功地找到了带目标键的元素。如果查找成功，名为 position 的输出参数将定位于表中的目标；如果查找失败，则函数的输出可能是一个未曾定义的值或者是在不同查找方法之间有所区别的值。

7.2 顺序查找

1. 算法和过程

毫无疑问，最简单的查找办法是在表的一端开始往后扫描，直到找到想要的键或者到达表的另一端，这是我们的第一种方法。

```
Error_code sequential_search(const List<Record> &the_list,
                             const Key &target, int &position)
/* Post: If an entry in the list has key equal to target, then return success and the
         output parameter position locates such an entry within the list.
         Otherwise return not_present and position becomes invalid. */
{
  int s=the_list.size();
  for(position=0; position<s; position ++){
    Record data;
    the_list.retrieve(position, data);
    if(data==target)return success;
  }
```

```
    return not_present;
}
```

这个函数中，只要在 Record 中没有找到键 target，for 循环就继续在表中往后移动，一旦找到了 target 就停止。如果查找不成功，则返回 not_present 值，并且在结束时 position 已移出了表的尾端(前面已提过，对不成功的查找，position 的值可能是未曾定义的)。

2. 分析

现在来估计顺序查找要做的工作量，以便与后面的其他查找技术进行比较。假设顺序查找在一个很长的表上运行。在 for 循环以外的语句仅执行一次，因此与循环内的工作相比所占的计算机时间无关紧要。在每一趟循环中，要将键与目标键进行比较，还要执行其他几个语句，并且检查几个表达式，但这些其他的语句和表达式与键的比较是以前后紧接的方式执行：它们在循环的每一趟都要做一遍。

因此要计算的所有操作与键的比较直接有关。如果另一个人使用同样的方法编写这个函数，则程序设计方法的区别可能造成运行时间有所区别，但所有这些情况仍然产生同样次数的键比较。如果表的长度发生变化，则这种查找方法的任何实现所做的工作也会成比例地变化。

我们要研究键比较次数对表长的依赖方式，这种研究能提供关于算法的最有用的信息。当真正编写程序时，无论决定使用哪种实现或程序设计技术，都可以同样地应用这些信息。

*** 比较次数的重要性

因此，如果我们要估计顺序查找可能需要的计算机时间，或者要将它与其他方法进行比较，那么知道它所做的键比较次数就给我们提供了最有用的信息，这些信息比全部运行时间实际上更为有用，因为运行时间太依赖于程序设计的变化和使用的特定机器。

对我们设计的无论哪种查找算法，我们能发表类似的声明，并将它作为分析查找算法的基本前提：算法所做的键比较次数反映了它的全部工作。

为了分析实施键比较的算法的性能，我们应该使用键比较次数作为算法所做工作量的度量标准。

当在 n 个元素的表上应用顺序查找时，它所进行的键比较次数是多少？由于顺序查找要将目标与表中的每个元素依次比较，因此答案依赖于是否有目标及目标在何处。如果函数在表的第一个位置就发现了目标，则它仅进行一次键比较；如果目标是第二个，则函数进行两次键比较；如果目标是表的最后一个，则函数要做 n 次键比较；如果查找不成功，则目标已经同表中的所有元素比较过，因而总共是 n 次键比较。

那么，依赖于是否有目标及在何处找到目标，我们的问题就有几个答案。如果查找不成功，则答案为 n 次键比较。成功查找的最佳性能是 1 次比较，最差为 n 次比较。

*** 平均性能

现在已得到了关于顺序查找的性能的十分详细的信息，这些信息对大多数用途来说实在是太详细了，因为通常我们不能确切地知道特定的键会出现在表中的哪个位置。代替地，如果能判断算法的平均性能，则通常会更有用。但平均对我们来说意味着什么？一个合理的假设就是将每种可能性各取一次并将结果进行平均，我们也总是这样假设。

* * * 附带条件

然而应该注意的是，这种假设可能会与实际情况相差甚远。例如，在一篇典型的文章中，并非所有的英语单词出现的次数都相等；电话接线员从一个大型企业接到的电话请求远比从一个普通家庭那儿接到的次数更多；C++编译器所遇到的关键词 if、class 和 return 也远比关键词 switch、continue 和 auto 多。

当算法选择的输入是按照某种统计分布时，在分析算法时就会伴有大量有趣的却又是极其困难的问题。然而，这些问题使我们离题太远，因而在此不予考虑。我们将注意力限于最重要的情况，即其中所有的概率都几乎相等的那一种情况。

在相等可能性的假设下，我们能发现成功的顺序查找中进行键比较的平均次数，只要将所有成功查找所需的键比较次数相加，再除以表中的元素数 n，结果是：

$$\frac{1+2+3+\cdots+n}{n}$$

在附录 A 中建立的第一个公式是：

$$1+2+3+\cdots+n=\frac{1}{2}n(n+1)$$

* * * 键比较的平均次数

因此顺序查找在成功情况下所做的键比较的平均次数是：

$$\frac{n(n+1)}{2n}=\frac{1}{2}(n+1)$$

3. 测试

对算法的理论分析的一种适当平衡是对结果函数的经验测试。我们建立样本数据，运行这个函数，并将结果与分析结果进行比较。

对查找函数，至少有两个数值得计算，即多遍查找所做的键比较的平均次数和要求的 CPU 时间量。现在设计一个函数，它能够用于测试我们的顺序查找例程。

为测试目的，我们使用整数键，且在记录中除了键以外不必再存储任何数据。在测试中，需要保存所有键比较操作的计数，一种方法是修改顺序查找函数，无论何时当它进行键比较时，都递增一个全局的计数器。然而，我们更愿意选择能避免对正被测试的函数进行任何改变的方法，改成通过修改重载的键比较操作来在调用键比较操作时递增计数器。这个计数器必须对所有的 Key 对象可用，因此它应当声明为静态(static)类成员。在 C++中，静态类成员提供的数据对象能被类的每个实例共享。① 因此，无论在哪里进行键比较，计数器的同一个实例 comparisons 将递增。

现在我们可以为测试程序完成下面的 Key 类定义。

```
class Key {
   int key;
public:
   static int comparisons;
   Key(int x=0);
```

① 见 C++课本中关于静态类成员的更全面的解释。

```
   int the_key() const;
   };
bool operator==(const Key &x, const Key &y);
bool operator> (const Key &x, const Key &y);
bool operator< (const Key &x, const Key &y);
bool operator>=(const Key &x, const Key &y);
bool operator<=(const Key &x, const Key &y);
bool operator!=(const Key &x, const Key &y);
```

我们使用方法 the_key 检查一个键的值的副本。对 Key 比较运算符的任何调用都会使静态计数器 comparisons 递增，例如，键的相等测试可如下实现：

```
bool operator==(const Key &x, const Key &y)
{
  Key::comparisons ++;
  return x.the_key()==y.the_key();
}
```

静态数据成员必须在类定义的外部定义并初始化，因此，将下列定义语句和类方法一起放在 Key 的实现文件 key.c 中。

```
int Key::comparisons=0;
```

由于我们的程序仅用于测试目的，没有必要让 Record 包含除 Key 以外更多的数据，因此定义：

```
typedef Key Record;
```

* * * 测试数据的选择

本章后面的大多数查找方法要求数据是排过序的，因此在测试函数中，使用键按升序排列的表。我们对成功的和非成功的查找都感兴趣，因此向表中仅插入包含奇数的键，然后通过寻找奇数进行成功的查找，而通过寻找偶数进行不成功的查找。如果表中有 n 个元素，那么成功查找的目标将是 $1,3,5,\cdots,2n-1$。对不成功的查找，我们寻找整数 $0,2,4,6,\cdots,2n$。这样我们就测试了所有可能的失败，包括目标小于表中的最小的键、在一对键之间以及大于最大的键。为了使测试更逼真，将应用附录 B 中的方法 Random::random_integer，使用伪随机数来选择目标。

* * * CPU 计时

在我们的测试中，使用附录 C 中的 Timer 类提供 CPU 计时信息。Timer 类的对象拥有的方法包括：启动计时器运行的构造函数和方法 reset、读取计时器的方法 elapsed_time。我们使用 Timer clock 首先对一组成功的查找进行计时，然后对类似次数的不成功查找进行计时，并由用户为参数 searches 提供一个值作为试验的次数。由此得到下面的测试函数：

```
void test_search(int searches, list<Record>&the_list)
/* Pre:  None
   Post: The number of key comparisons and CPU time for a sequential searching function
         have been calculated.
   Uses: Methods of the classes List, Random, and Timer, together with an output
         function print_out*/
{
   int list_size=the_list.size();
```

```
  if (searches)<=0||list_size<0){
     cout <<"Exiting test:"<<endl
          <<"The number of searches must be positive."<<endl
          <<"The number of list entries must exceed 0."<<endl;
     return
  }
  int i, target, found_at;
  Key::comparisons=0;
  Random number;
  Timer clock;
  for(i=0;i<searches; i++){
    target=2*number.random_integer(0,list_size-1)+ 1;
    if (sequential_search(the_list, target, found_at)==not_present)
      cout<<"Error:Failed to find expected target"<<target<<endl;
  }
  print_out("Successful", clock.elapsed_time(),Key::comparisons,searches);
  Key::comparisons=0
  clock,reset();
  for(i=0;i<searches;i++){
    target=2*number.random_integer(0,list_size);
    if (sequential_search(the_list,target,found_at)==success)
      count <<"Error:Found unexpected target"<<target
            <<"at"<<found_at<<endl;
  }
  print_out("Unsuccessful",clock.elapsed_time(),Key::comparisons,searches);
}
```

将这个函数嵌入一个能运行的程序以及编写输出函数 print_out 的细节将留作一个项目。

练习 7.2

E1. 对任何算法的一种有效的检查是了解它在极端情况下做什么。判断当下列情况发生时，顺序查找做什么。

(1)表中仅有一个元素。

(2)表是空的。

(3)表是满的。

E2. 一个表包含 3 个元素，跟踪顺序查找对表中每个键的查找。判断发生了多少次比较，并因此检查成功查找的平均比较次数的公式。

E3. 如果能够假设表中的键已按顺序排列(例如数字或字母顺序)，那么就能更快地终止不成功的查找。如果最小的键在最前面，则一旦一个键大于或等于被查找的目标键，就可以终止查找。如果假设不在表中的目标键落入 $n+1$ 个区间(在第一个键之前、在一对相继的键之间、在最后一个键之后)的概率相等，那么在此版本中不成功的查找的平均比较次数是多少?

* * * 监视哨

E4. 顺序查找在每次重复时要检查两个不等式，其中一个是键的比较，用以查看是否已找到目标 target，另一个是下标的比较，用以查看是否到达表的尾部。通过消除第二个比较去加速算法的好办法是增加表的长度并在表尾插入一个附加的元素，它的键为 target，

从而确保最终总能找到键 target。在表中放入的保证一个过程终止的元素称为监视哨(sentinel)。当循环终止时,如果在表中最后一个元素前发现 target,则查找成功,否则,如果监视哨正好是所发现的 target,则查找不成功。

编写一个 C++函数,使用 6.2.2 节中设计的表,在顺序版本的顺序查找中体现出这种思想。

E5. 找出下列条件下,练习 E4 中编写的函数所做的键比较的次数。

(1)不成功的查找。

(2)最佳的成功查找。

(3)最差的成功查找。

(4)平均的成功查找。

程序设计项目 7.2

P1. 编写一个程序,使用 6.2.2 节中设计的表去测试顺序查找以及后面的其他查找方法。做出适当的声明来创建这个表并将键放入表中,这些键是 1~n 的奇数,其中 n 是用户给出的值。因此成功的查找可以通过寻找奇数进行测试,而不成功的查找则可以通过寻找偶数进行测试。使用课本中的 test_search 对查找函数进行真实的测试。重载键比较运算符以使它们递增计数器 counter。编写合适的 introduction 函数、print_out 函数及菜单驱动程序。目前惟一的选项是以用户给定数目的元素填充这个表,测试 sequential_search 函数并退出。以后可以加入其他的查找函数作为更多的选项。

找出在不成功的和成功的查找这两种情况下所做的比较次数,并将这些结果与课本中的分析进行比较。

用具有代表性的 n 值运行你的程序,如 $n=10$,$n=100$,$n=1000$。

* * * 监视哨查找

P2. 利用项目 P1 中编写的驱动程序去测试查找函数,并插入使用监视哨的顺序查找版本(见练习 E4)。对 n 的各种值,判断是带监视哨还是不带监视哨的版本更快。通过实验,如果两个版本之间有交叉点的话,找出这个交叉点。也就是说,当 n 为何值时,在长度为 n 的表的最后插入监视哨所需的额外时间与不带监视哨的版本中额外的下标比较所需的时间一样?

* * * 链式顺序查找

P3. 为了在如 6.2.3 节设计的简单的链表上操作,需要对我们的顺序查找函数和测试程序进行哪些修改?做出这些修改,并将项目 P1 中的测试程序应用于链表,以测试链式顺序查找。

7.3 二分查找

顺序查找易于编写,并且对较短的表有一定的效率,但对较长的表则是灾难。例如想像要在厚厚的电话本中从头开始每次读出一个名字以找到"Amanda Thompson"这个名字。为了在一个较长的表中找出任何元素,倘若这个表中的键已经排过序,则还有更多的有效的

方法。

* * * 方法

用于键有序的表的最好的方法之一是首先将目标键与表中央的键进行比较，然后根据目标键是在中央这个键的前面还是后面，将注意力仅仅限制到表的第一部分或第二部分。通过一次键比较就可以将表长减少到其原来长度的一半。在每一步都继续这一操作，则每次都可以将要查找的表长减少到一半。仅仅在 20 步里，这种方法就可以在包含了上百万个键的表中查找到任何要求的键。

* * * 限制

我们讨论的这种方法称为二分查找。这种方法当然要求表中的键是标量或者是其他有序的类型，并且这个表已经是完全有序的。

7.3.1 有序表

我们这时真正要做的是引入一个新的抽象数据类型，它以下面的方式定义。

定义 有序表是一种表，表中的每个元素包含一个键，这些键是有序的。即，如果表中元素 i 在元素 j 之前，则元素 i 的键小于或等于元素 j 的键。

如果不加修改，则对有序表惟一不适用的 List 操作是 insert 和 replace。这些标准的 List 操作肯定失败，否则，它们会打乱表的顺序。因此我们将有序表作为从顺序 List 派生的类来实现，在这个派生类里，我们将用新的实现重载 insert 方法和 replace 方法。因此，使用下面的类说明：

```
class Ordered_list: public List<Record> {
public:
  Ordered_list();
  Error_code insert(const Record &data);
  Error_code insert(int position, const Record &data);
  Error_code replace(int position, const Record &data);
};
```

我们重载 insert 方法和 replace 方法，并重载了另一个 insert 方法，使得能用单个参数去使用它，这个被重载的方法将一个元素放入由键的顺序所决定的正确的位置。我们将在第 8 章进一步学习这种操作，这里只是重载方法的一个简单的、独立于实现的版本。

如果表中已包含了与待插入的键相等的键，则新的键将作为所有与它相等的键的第一个插入。

```
Error_code Ordered_list::insert(const Record &data)
/* Post: If the Ordered_list is not full, the function succeeds: The Record data is
        inserted into the list, following the last entry of the list with a strictly
        lesser key (or in the first list position if no list element has a lesser key).
        Else: the function fails with the diagnostic Error_code overflow.*/
{
  int s=size();
  int position;
  for(position=0; position< s; position++){
```

```
    Record list_data;
    retrieve(position, list_data);
    if(data>=list_data)break;
  }
  return List<Record> ::insert(position, data);
}
```

* * * 作用域解析

这里通过使用作用域解析运算符去应用基类 List 最初的 insert 方法，这个作用域解析运算符是必要的，因为我们已经用新的 Ordered_list 方法覆盖了原来的插入方法，Ordered_list 的 insert 方法的编码如下：

```
Error_code Ordered_list::insert(int position, const Record &data)
/* Post: If the Ordered_list is not full, 0≤position≤n, where n is the number of
        entries in the list, and the Record data can be inserted at position in the
        list, without disturbing the list order, then the function succeeds: Any entry
        formerly in position and all later entries have their position numbers
        increased by 1 and data is inserted at position of the list.
        Else: the function fails with a diagnostic Error_code.*/
{
  Record list_data;
  if (position> 0){
    retrieve(position -1, list_data);
    if (data< list_data)
      return fail;
  }
  if (position< size()){
    retrieve(position, list_data);
    if (data> list_data)
      return fail;
  }
  return List< Record> ::insert(position, data);
}
```

注意派生类中覆盖和重载方法的区别：覆盖方法是用具有相同的名字和参数列表的方法来代替基类的方法，而重载方法仅是与已有的方法具有相同的名字但却有不同的参数列表。

7.3.2 算法设计

* * * 危险性

二分查找的思想虽然简单，但在对它进行程序设计时却非常容易出错。此方法至少可以追溯到 1946 年，但第一个没有错误和没有不必要限制的版本是出现在 1962 年。一项研究（见本书后面的参考书目）表明约有 90%的职业程序设计员即使在干了整整一小时后也不能正确地编出二分查找。另一项研究[①]发现，20 本课本中仅有 5 本给出了二分查找的正确

① Richard E. Pattis, "Textbook errors in binary searching," *SIGCSE Bulletin*, 20 (1988), 190—194.

的解决方法。

因此我们必须特别谨慎，确信我们自己没有犯错。为此必须准确地声明我们的变量指派了什么，必须精确地规定在程序中包含的循环在每次重复前后什么条件必须为真，必须确信循环会正确地终止。

我们的二分查找算法将使用两个下标 top 和 bottom，它们围住表中我们正在寻找目标键的那一部分。在每次重复时，二分查找会将这部分表的长度减少大约一半。为帮助记录算法的进展，我们在此过程的每次重复前写下一条要求它为真的断言，这样的语句称为此过程的不变式(invariant)。

* * * 不变式

假定目标键在表中出现，则目标键将在下标 bottom 和 top 之间的范围内被找到。

我们将 bottom 置为 0，并将 top 置为 the_list. size() - 1 来建立这个断言的初始正确性。

为了进行二分查找，首先计算 bottom 和 top 之间的中间位置 mid：

```
mid= (bottom+ top) /2
```

接下来，将目标键与 mid 位置上的键进行比较，然后修改下标 top 或 bottom 中适当的一个，将表减少到 bottom 或者 top 所在的那一半。

* * * 终止

再接下来，我们注意到，假定在前面都没有因找到目标而终止的话，则二分查找会在 top≤bottom 时终止，即当表的剩余部分至多包含一项时终止。

* * * 前进

最后，必须保证剩下的待查找的表项数 top - bottom + 1 在此过程中的每次重复后都会绝对地减少，从而保证此过程向着终止前进。

可以编写几个稍微不同的二分查找算法。

7.3.3 健忘版

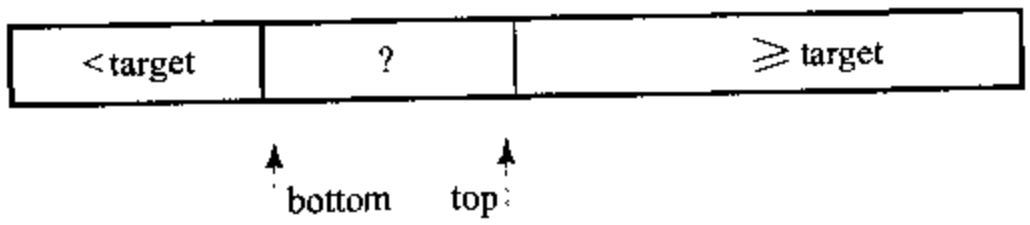
或许最简单的变化是忘记键目标 Key target 可能很快被找到，而不管是否已经找到 target，查找算法都继续对表进行再分，直到剩下的表的长度为 1。

下面的函数实现了这种方法，其中为了程序设计的简单性，采用递归的形式编写，将子表的边界作为递归函数的一个附加的参数给出。

```
Error_code recursive_binary_1(const Ordered_list &the_list, const Key &target,
                              int bottom, int top, int &position)
/* Pre:  The indices bottom tp top define the range in the list to search for the target.
   Post: If a Record in the range of locations from bottom to top in the_list has key
         equal to target, then position locates one such entry and a code of success is
         returned. Otherwise, the Error_code of not_present is returned and position
         becomes undefined.
   Uses: recursive_binary_1 and methods of the classes List and Record. */
{
   Record data;
   if (bottom< top) {          //  List has more than one entry.
      int mid= (bottom+ top) /2;
```

```
    the_list.retrieve(mid, data);
    if (data< target)       //  Reduce to top half of list.
      return recursive_binary_1(the_list, target,mid+1, top, position);
    else                    //  Reduce to bottom half of list.
      return recursive_binary_1(the_list, target,bottom, mid, position);
  }
  else if(top<bottom)
    return not_present;     //  List is empty.
  else {                    //  List has exactly one entry.
    position=bottom;
    the_list.retrieve(bottom, data);
    if(data== target)return success;
    else return not_present;
  }
}
```

下面的图描述了表到子表的分割：



注意，此图表明仅严格小于 target 的元素在表的开头部分中，而表的末尾部分中包含的元素都大于或等于 target。这样，如果在表中目标出现超过一次，则当表的中间部分长度减为 1 并且命中目标时，可以保证它是第一个出现的目标。

如果这个表是空的，则函数失败，否则它首先计算 mid 的值，mid 作为 bottom 和 top 的平均值，位于两者之间，因此 mid 必是表中某个合法元素的下标。

* * * 终止

注意，调用递归的 if 语句不是对称的，因为测试的条件将 mid 放入两个区间中较低的一个。另一方面，非负整数的整除总是向下截取。这两个条件共同保证递归总会终止。让我们判断查找要结束时将发生什么。只要 top>bottom，递归就会继续。但这个条件意味着在计算 mid 时，由于整除是向下截取，因此我们总有 bottom<=mid<top。

接下来，if 语句将区间的长度从 top-bottom 减至 top-(mid+1)或者是 mid-bottom，由上面这个不等式，这两个长度中的任一个都绝对小于 top-bottom，因此每次重复时，此区间的长度绝对地减少，因此递归最终将终止。

在递归终止后，我们必须首先检查是否找到了目标键，因为所有以前的比较都只是在测试不等式。

为了按照我们关于标准查找函数的约定调整参数，我们得到下面的查找函数：

* * * 对 recursive_binary1 的主调用

```
Error_code run_recursive_binary_1(const Ordered_list &the_list,
                                  const Key &target, int &position)

{
return recursive_binary_1(the_list, target, 0, the_list.size()-1, position);
}
```

由于函数 recursive_binary_1 中使用的递归是尾部递归，因此能够轻易地将它转换成一

个重复循环。同时，也能够使参数与其他查找方法一致。

```
Error_code binary_search_1(const Ordered_list &the_list,
                           const Key &target, int &position)
/* Post: If a Record in the list has Key equal to target, then position locates one such
         entry and a code of success is returned. Otherwise, not_present is returned and
         position is undefined.
   Uses: Methods for classes List and Record. */
{
   Record data;
   int bottom=0, top=the_list.size()-1;
   while(bottom< top){
    int mid=(bottom+ top)/2;
    the_list.retrieve(mid, data);
    if (data< target)
       bottom=mid+1;
    else
       top=mid;
   }
   if(top< bottom) return not_present;
   else {
      position=bottom;
      the_list.retrieve(bottom, data);
      if(data==target) return success;
      else return not_present;
   }
}
```

7.3.4 识别相等

虽然 binary_search_1 是一种简单形式的二分查找，但看起来它经常做些不必要的重复，因为它不能在继续重复前识别已找到了目标。因此我们可能希望使用一个变量以在每个阶段进行检查，了解是否已发现了目标，从而节省计算机时间。

采用递归形式后，这种方法变为：

```
Error_code recursive_binary_2(const Ordered_list &the_list, const Key &target,
                              int bottom, int top, int &position)
/* Pre:  The indices bottom to top define the range in the list to search for the target.
   Post: If a Record in the range from bottom to top in the_list has key equal to target,
         then position locates one such entry, and a code of success is returned.
         Otherwise, not_present is returned, and position is undefined.
   Uses: recursive_binary_2, together with methods from the classes Ordered_list and
         Record. */
{
   Record data;
   if (bottom<=top) {
      int mid=(bottom + top)/2;
      the_list.retrieve(mid, data);
      if (data==target) {
         position=mid;
         return success;
      }
```

```
    else if(data< target)
      return recursive_binary_2(the_list, target, mid + 1, top, position);
    else
      return recursive_binary_2(the_list, target, bottom, mid-1, position);
  }
  else return not_present;
}
```

同 run_recursive_binary_1 一样，我们需要函数 run_recursive_binary_2 来按我们的标准约定调整参数。

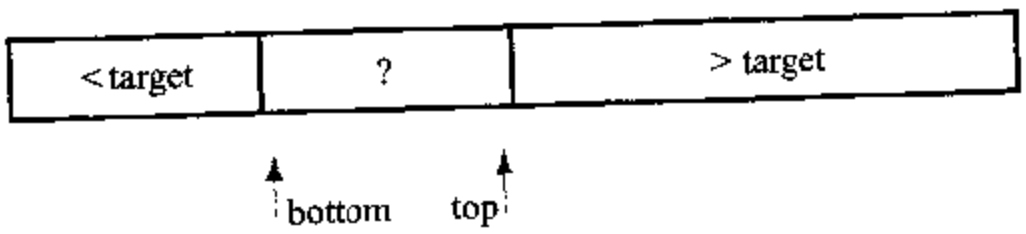
* * * 到 recursive_binary_2 的主调用

```
Error_code run_recursive_binary_2(const Ordered_list &the_list,
                                  const Key &target, int &position)
{
return recursive_binary_2(the_list, target, 0, the_list.size()-1, position);
}
```

同样，这个函数也能转换成仅带标准参数的非递归形式：

```
Error_code binary_search_2(const Ordered_list &the_list,
                           const Key &target, int &position)
/* Post: If a Record in the_list has key equal to target, then position locates one such
         entry and a code of success is returned. Otherwise, not_present is returned and
         position is undefined.
   Uses: Methods for classes Ordered_list and Record.*/
{
  Record data;
  int bottom=0, top=the_list.size()-1;
  while(bottom<=top){
   position=(bottom + top)/2;
   the_list.retrieve(position, data);
   if(data==target) return success;
   if(data< target)bottom=position + 1;
   else top=position -1;
  }
  return not_present;
}
```

在下图中描述了这种版本的操作：



注意，此图(与第一种方法对比)是对称的，它的开头部分只包含了严格小于 target 的元素，而末尾部分只包含了严格大于 target 的元素。因此，用这种方法，如果 target 在表中出现超过一次，则算法会返回目标的任何一个实例。

* * * 循环终止

证明在 binary_search_2 中循环会终止比对 binary_search_1 的证明更简单。在 binary_search_2 中，循环中的 if 语句的形式保证区间的长度在每次重复后会至少减少一半。

* * * 方法的比较

这两个版本的二分查找中哪一个进行的键比较更少？如果我们碰巧在接近查找开始的地方就找到了目标，则答案显然是 binary_search_2。但 binary_search_2 的每次重复需要两次键比较，而 binary_search_1 只要一次。是否可能在许多次重复后，使得 binary_search_1 可以进行较少的键比较？为了回答这个问题，我们将在下一节设计新的分析工具。

练习 7.3

E1. 假设 the_list 包含整数 1,2,…,8。跟踪 binary_search_1 的步骤，判断它在查找下列每个目标时各做了多少次键比较：(1)3；(2)5；(3)1；(4)9；(5)4.5。

E2. 用 binary_search_2 重复练习 E1。

E3. (挑战)假设 L_1 和 L_2 是有序表，分别包含 n_1 和 n_2 个整数。

(1)使用二分查找的思想描述如何在组合表中找出 n_1+n_2 个整数的中间数字。

(2)编写函数实现你的算法。

程序设计项目 7.3

P1. 利用 7.2 节项目 P1 的驱动程序，将 binary_search_1 和 binary_search_2 作为查找选项。比较这两个算法的性能，并将它们与顺序查找进行性能比较。

P2. 在 7.2 节项目 P1 的测试程序中并入二分查找的递归版本(两个变种)，将它们与非递归的二分查找进行性能比较。

7.4 比较树

* * * 定义

比较树(comparison tree)，又称为决策树(decision tree)或查找树(search tree)，它通过从头到尾跟踪算法的动作而得到，它将每次键比较表达成树的一个顶点(将它画成一个圆圈)。在圆圈中放入与目标键进行比较的键的下标。从圆圈画出的分支(线)表示可能的比较结果并据此标注分支。当算法终止时，我们将 F(失败)或者是找到的目标的位置放在适当的分支的末端，称之为树叶，并画作一个方格。树叶有时也称为树的端顶点或外部顶点，其余顶点称为树的内部顶点。

顺序查找的比较树特别简单，如图 7.2 所示。

* * * 定义

在一次特定的查找中算法所进行的比较次数是从树顶(称为根)向下直到树叶的路径上遍历的内部(用圆圈表示)顶点数。从根到达一个顶点所遍历的分支称为此顶点的层(level)。因此，根在 0 层，直接在根下面的顶点在 1 层，以此类推。

最长路径中的顶点数称为树的高度，因此仅有一个顶点的树的高度为 1。在以后的章节中，我们有时也允许树是空的(即完全不包含任何顶点)，并且约定空树的高度为 0。

现在按惯例将族谱和植物树混合比喻来定义用于树的术语：我们称直接在顶点 v 下面

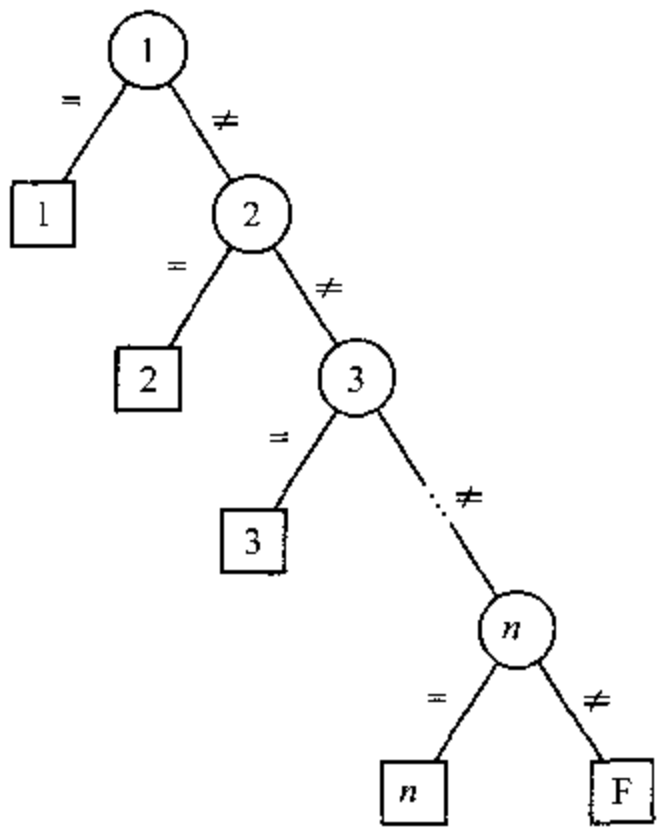


图 7.2　顺序查找的比较树

的顶点为 v 的孩子，直接在顶点 v 之上的顶点为 v 的双亲。因此可以使用诸如“树叶的双亲”或“树根的孩子”这样的矛盾修饰法。

7.4.1　$n = 10$ 时的分析

1. 树的形状

图 7.3 和图 7.4 分别画出了 binary_search_1 和 binary_search_2 的比较树，从顺序查找的树的形状同这两棵二分查找的树的形状的比较中，明显可以看出顺序查找平均所做的比较远远多于二分查找。顺序查找有一棵狭长的树，它意味着许多比较，而二分查找的树更宽

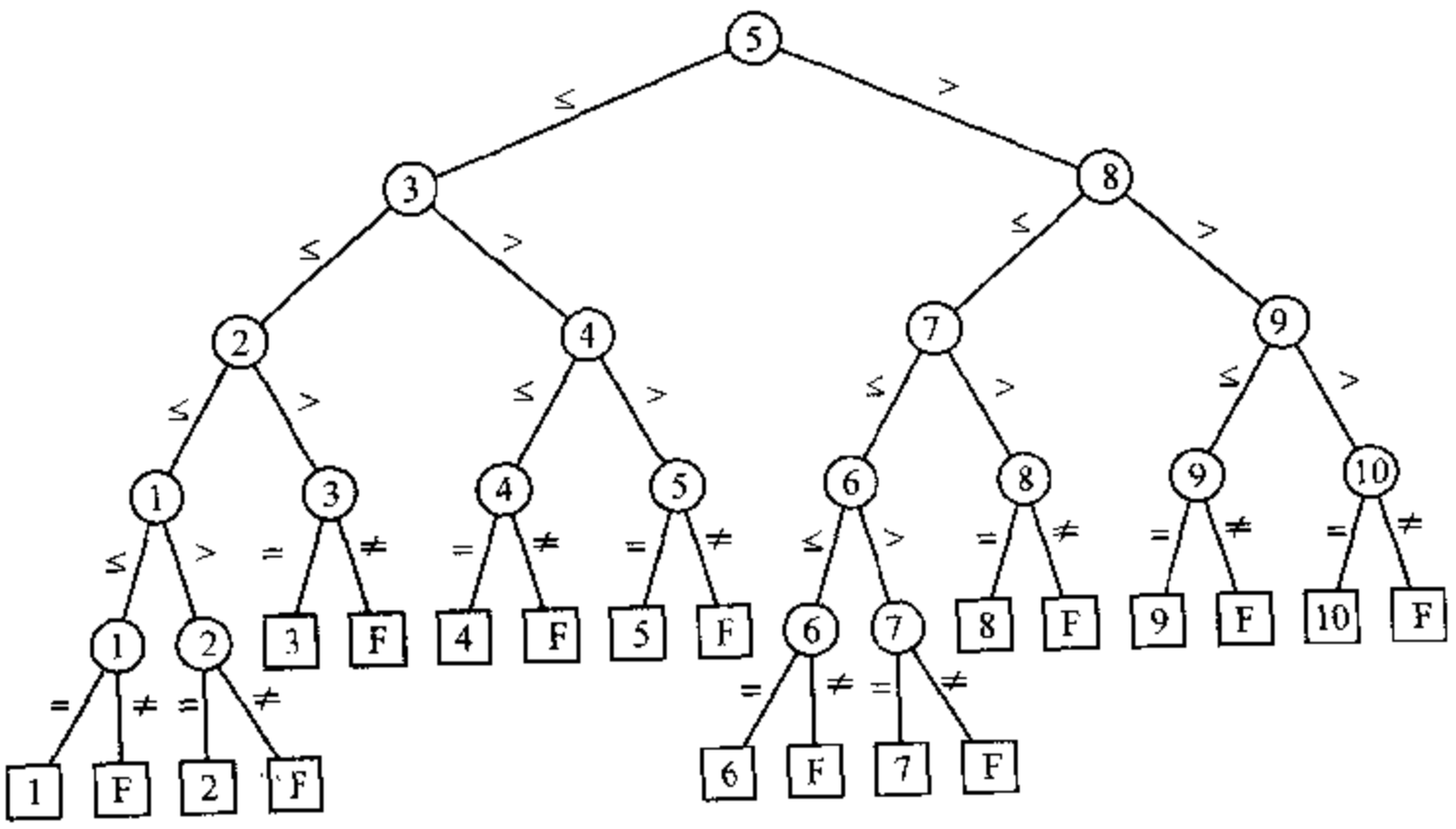


图 7.3　$n = 10$ 时 binary_search_1 的比较树

且更矮。

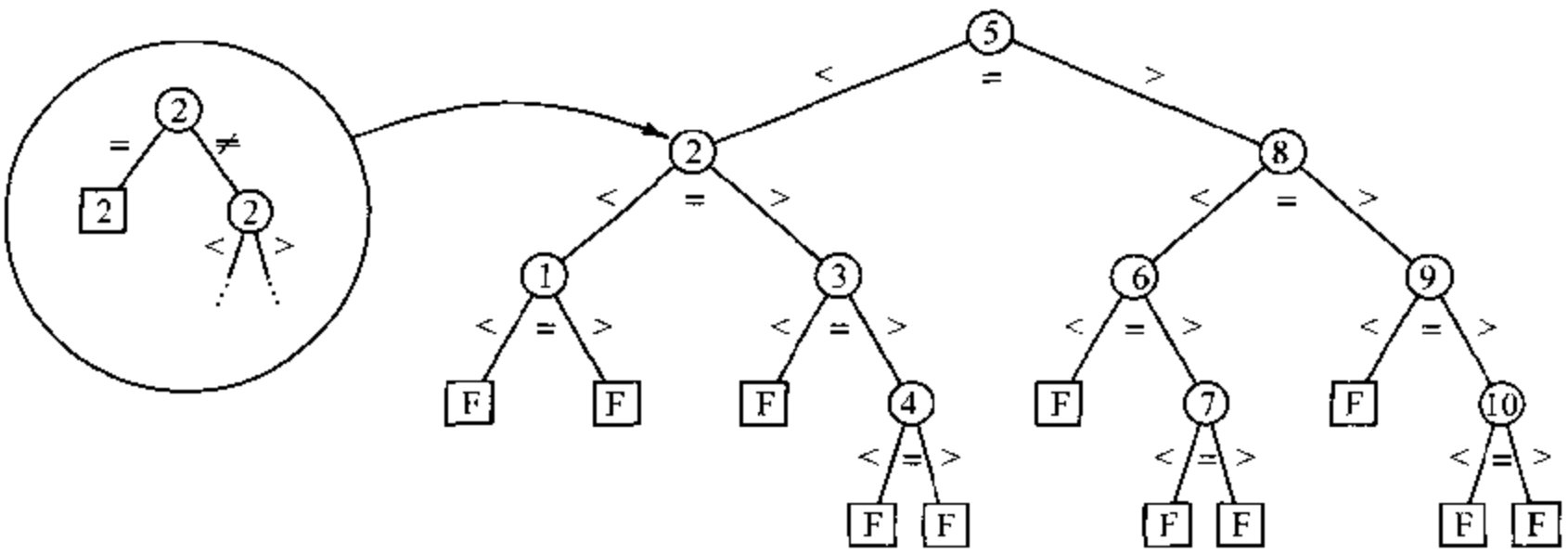


图 7.4 $n = 10$ 时 binary_search_2 的比较树

2. 三路比较和紧凑的图

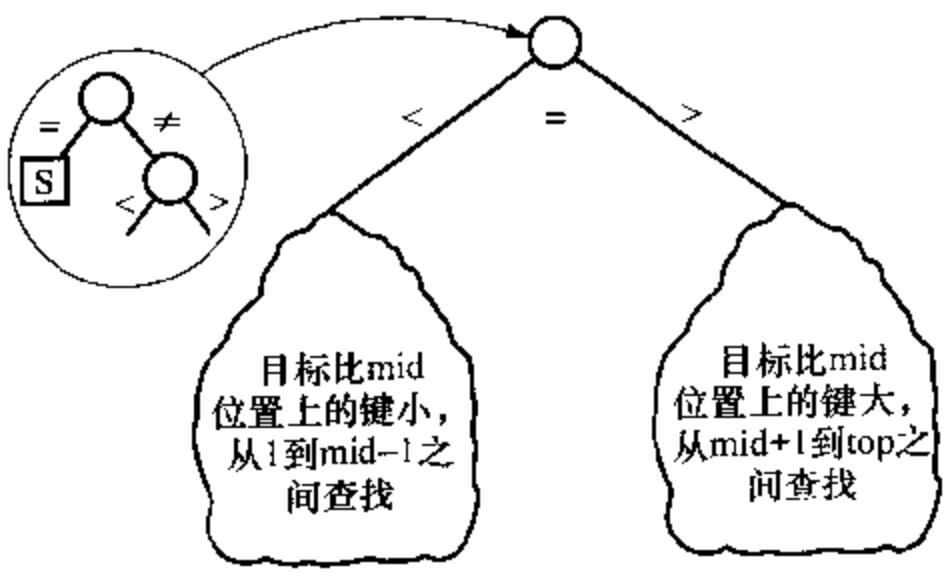
*** 扩展的和浓缩的树

在为 binary_search_2 画出的树中，我们通过在循环的每一遍中将两次比较结合成为一次三路比较，从而更清晰地显示了算法的结构(并减少了所需的空间)。以此方式画出的树意味着每个非树叶的顶点将终止一些成功的查找，而树叶则对应于不成功的查找。因此图 7.4 更加紧凑，但记住对所示的每个顶点，除了在成功找到目标的顶点上仅进行一次比较以外，在其他顶点实际上都进行了两次比较。

正是这种画比较树的紧凑方法将成为将来的章节中的标准方法。

*** 图解树

它也经常便于只显示比较树的一部分。图 7.5 显示了递归版本的 binary_search_2 的比较树的顶部，递归调用的所有细节都隐藏在子树中。递归算法的比较树和递归树经常是考虑同一问题的两种方式。



目标比mid位置上的键小，从1到mid−1之间查找

目标比mid位置上的键大，从mid+1到top之间查找

图 7.5 递归的 binary_search_2 的比较树的顶部

从所给出的 $n=10$ 时的 binary_search_1 和 binary_search_2 的树中，不难看出每种算法将要进行多少次比较。在最坏情况下的查找中，这个数字只比树的高度大 1。实际上，对每

次查找，这个数字是根和查找终止顶点之间存在的内部顶点的数目。

3. binary_search_1 的比较次数

* * * 外部路径长度

在 binary_search_1 中，每次查找都终止于一个树叶。为了获得成功查找和不成功查找两者的平均比较次数，我们需要所谓的树的外部路径长度(external path length)：即从根到树中每个树叶各遍历一次所经过的分支数的总和。对图 7.3 中的树，外部路径的长度为：

$$(4\times5)+(6\times4)+(4\times5)+(6\times4)=88$$

其中一半树叶对应于成功的查找，另一半则对应于不成功的查找。因此当 $n=10$ 时，binary_search_1 的成功的或者不成功的查找所需的平均比较次数为 44/10=4.4。

4. binary_search_2 的比较次数

* * * 内部路径长度

在为 binary_search_2 画出的树中，所有的树叶对应于不成功的查找，因此外部路径长度即为不成功的比较次数。对成功的查找，我们需要内部路径长度(internal path length)，它定义为从根到所有非树叶上的分支数的和。对图 7.4 中的树，内部路径长度为：

$$0+1+2+2+3+1+2+3+2+3=19$$

* * * 平均成功次数

回忆一下，binary_scarch_2 除了找到目标顶点之外，要为每个非树叶各进行两次比较，并且注意所遍历的这些内部顶点数比分支数多 1(对 $n=10$ 时的每个顶点)。因此我们得到成功查找的平均比较次数为：

$$2\times(\frac{19}{10}+1)-1=4.8$$

上式中减去 1 是对应于当目标找到时少进行 1 次比较的事实。

对 binary_search_2 的不成功的查找，我们需要图 7.4 中的树的外部路径长度。它是：

$$(5\times3)+(6\times4)=39$$

* * * 平均的不成功次数

对不成功的查找，假设 $n+1$ 个区间(小于第一个键，在一对相继的键之间，或者大于最大的键)都是相等的查找概率，则对于图中可假设 11 个失败的树叶有相同的查找概率。因此一次不成功查找的平均比较次数是：

$$\frac{2\times39}{11}\approx7.1$$

5. 算法的比较

对 $n=10$ 的情况，binary_search_1 对成功和不成功的查找都进行略少的比较。然而为公平起见，我们应该指出 binary_search_2 在每个内部顶点处所进行的两次比较是紧密相关的(同样的键被比较)，因此优化的编译器可能不会做与两次完整比较相同的工作。实际上，在那种情况下，当 $n=10$ 时，对成功的查找，binary_search_2 可能比 binary_search_1 是略好的选择。

7.4.2 一般化

当 n 大于 10 时会发生什么情况？对较长的表，不可能画出整个比较树，但从 $n=10$ 的例子中，我们可以做出一些始终是正确的观察结论。

1. 2-树

我们将 2-树定义成这样的树，即除树叶外每个顶点都正好有两个孩子。我们所画的两个版本的比较树都适合此定义，因此是 2-树。我们可以做出几个有关 2-树的结论，它们将提供有关所有 n 值的二分查找方法的性能的信息。

* * * 术语

2-树的其他术语是严格的二叉树(strictly binary tree)和扩展的二叉树(extended binary tree)，我们不应使用这些术语，因为它们太容易与二叉树混淆，二叉树(将在第 10 章介绍)有不同的意义。

* * * 2-树中的顶点数

在 2-树中，任一层上的顶点数不能超过上一层顶点数的 2 倍，因为每个顶点都只有 0 或 2 个孩子(依赖于它是否是树叶)。因为在第 0 层只有一个顶点(根)，则对所有的 $t\geqslant 0$，第 t 层的顶点数最多为 2^t。因此有如下事实：

引理 7.1 2-树的每一层上的顶点数最多是其直接上层中的顶点数的 2 倍。因此，在 2-树中，对 $t\geqslant 0$，第 t 层的顶点数最多为 2^t。

如果愿意的话，可以将后一个观察数据用对数进行转换。假设在 t 层有 k 个顶点，(由引理 7.1 的第二部分)因为 $k\leqslant 2^t$，我们得到 $t\geqslant \lg k$，这里的 lg 表示以 2 为底数的对数。①

引理 7.2 如果 2-树在第 t 层有 k 个顶点，则 $t\geqslant \lg k$，这里的 lg 表示以 2 为底数的对数。

* * * 对数

刚才使用的以 2 为底数的对数的符号将是本书通篇所用的标准符号。在分析算法时，有时也需要自然对数(以 e=2.71828...为底)，我们将自然对数表示为 ln。我们很少使用其他底数的对数，因此总结如下：

> 约定
>
> 除非另外说明，所有的对数都将以 2 为底数。
>
> 符号 lg 表示以 2 为底数的对数，而符号 ln 表示自然对数。
>
> 当对数的底数没有指定时(或不重要时)，则将使用 log 符号。

* * * 下限和上限

当采用对数后，经常需要向上或向下移到下一个整数。为说明此操作，我们将一个实数

① 对数性质的复习见附录 A。

x 的下限定义为小于或等于 x 的最大整数，将 x 的上限定义为大于或等于 x 的最小整数。我们用 $\lfloor x \rfloor$ 表示 x 的下限，而用 $\lceil x \rceil$ 表示 x 的上限。

对整数 n，注意：

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$
$$(n-1)/2 \leqslant \lfloor n/2 \rfloor \leqslant n/2$$
$$n/2 \leqslant \lceil n/2 \rceil \leqslant (n+1)/2$$

2. 关于 binary_search_1 的分析

我们现在开始进行 n 个元素的表的 binary_search_1 的全面分析。在 binary_search_1 中所做的最后步骤通常是目标相等性检查，因此成功的和非成功的查找都在树叶处终止，因此总共有正好 $2n$ 个树叶。如图 7.3 中对 $n=10$ 的说明，所有这些树叶必须在同一层或在相邻的两层上。（这个观察可以用数学归纳法建立下列更强的命题进行证明：如果 T_1 和 T_2 是在表 L_1 和 L_2 上运行的 binary_search_1 的比较树，且 L_1 和 L_2 的长度最多相差 1，则 T_1 和 T_2 的所有树叶是在相同的或相邻的层上。当 L_1 和 L_2 是长度至多为 2 的表时，这一命题完全正确。另外，如果 binary_search_1 分割两个长度相差最多为 1 的较大的表，得到的 4 半的长度也最多相差 1，则归纳假设表明它们的树叶都在相同的或相邻的层上。）从引理 7.2 得到，比较树中最大的树叶层 t 满足 $t=\lceil \lg 2n \rceil$。

由于在根（它是第 0 层）上进行一次键比较，但在树叶（第 t 层）上不进行比较，随之得到键比较的最大次数也是 $t=\lceil \lg 2n \rceil$。因为所有的树叶都在相同的或相邻的层上，所以这个最大次数最多只比平均数目大 1。

因此我们有：

* * * binary_search_1 的比较次数

binary_search_1 在查找 n 个元素的表时所进行的键比较次数在最坏情况下大约是 $\lg n+1$，在平均情况下大约是 $\lg n$。比较次数实质上独立于查找成功与否。

3. 关于 binary_search_2 不成功的查找的分析

* * * binary_search_2 不成功情况下的比较次数

为了计算 binary_search_2 对通用 n 值进行不成功查找时所进行的比较次数，需要检查它的比较树。与 binary_search_1 中所给出的那些原因类似，此树同样在顶部是满的，它的所有树叶最多是在底部的两个相邻层上。对 binary_search_2 来说，所有的树叶对应于不成功的查找，因此正好有 $n+1$ 个树叶，对应 $n+1$ 种不成功的结果：比最小的键小，在一对相继的键之间以及比最大的键大。由于这些树叶都在树的底部，引理 7.1 指出树叶的数目大约是 2^h，这里 h 是树的高度。利用（底数为 2 的）对数，我们得到 $h \approx \lg(n+1)$。这个值是从根到任一个树叶的近似距离。由于在 binary_search_2 中，要对每个内部顶点执行两次比较，因此在不成功的查找中所进行的比较次数大约是 $2\lg(n+1)$。

binary_search_2 在不成功的查找中所进行的比较次数大约是 $2\lg(n+1)$。

4. 通路长度定理

为了计算 binary_search_2 成功查找的平均比较次数，我们首先获得一个对任何 2-树都成立的有趣而重要的关系式。

定理 7.3 用 E 表示一棵 2-树的外部路径长度，用 I 表示内部路径长度，q 为非树叶的顶点数目，则 $E=I+2q$。

证明 为了证明这个定理，我们使用数学归纳法，对树中的顶点数做归纳。

如果树中仅包含根且没有其他顶点，则 $E=I=q=0$，则此定理的基础条件一般正确。

现在取一棵较大的树，设 v 是非树叶，但 v 的两个孩子均是树叶。设 k 是从根到 v 的路径上的分支数，见图 7.6。

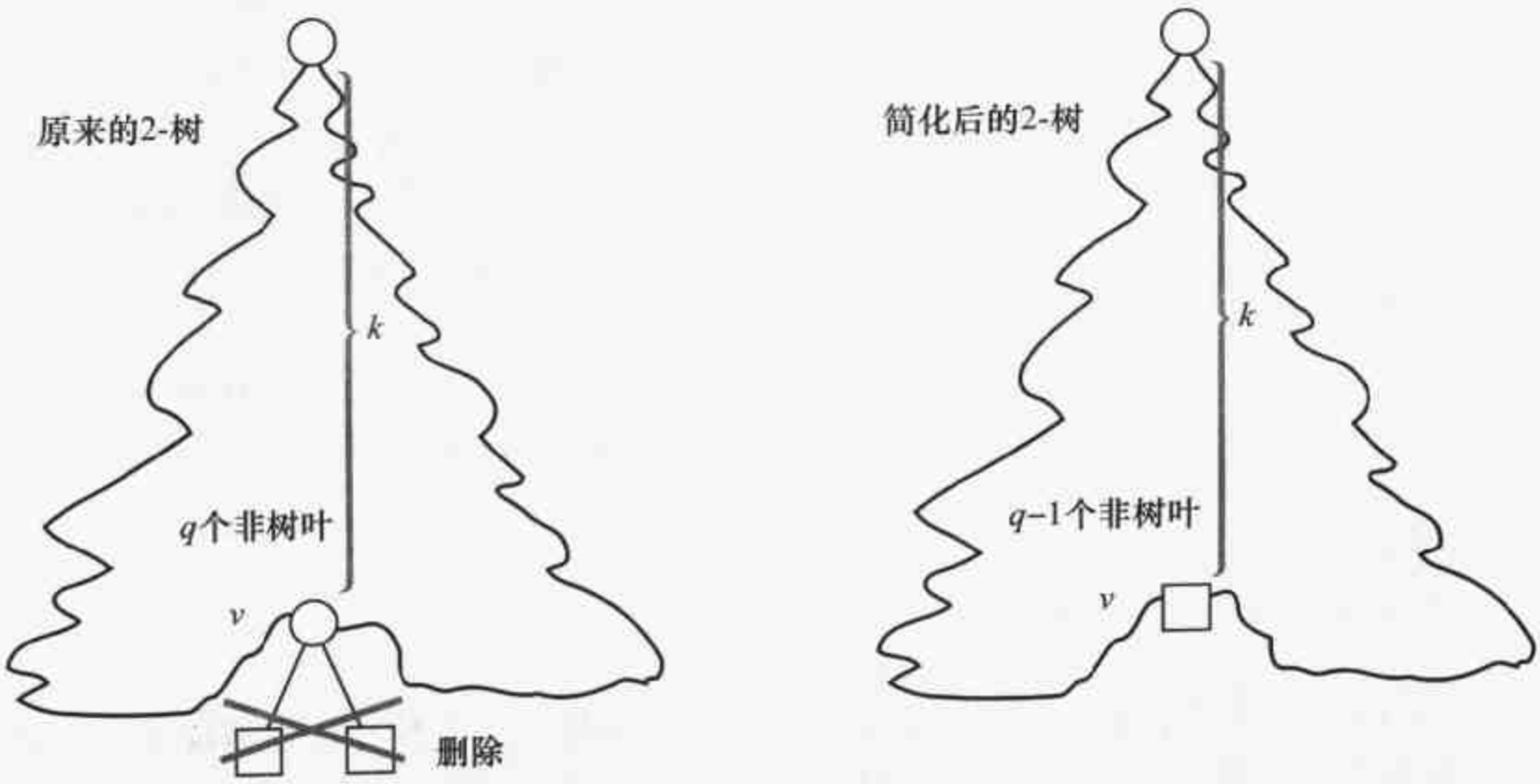


图 7.6 2-树中的通路长度

现在我们从 2-树中删除 v 的两个孩子。由于 v 不是树叶，但它的孩子却是树叶，则非树叶的数目从 q 个降为 $q-1$ 个。内部路径长度 I 要减去到 v 的距离，即变成 $I-k$。到 v 的每个孩子的距离是 $k+1$，因此外部路径长度从 E 减为 $E-2(k+1)$，但由于 v 现在变成了树叶，因此必须加上它的距离 k，这样就得到新的外部路径长度为：

$$E-2(k+1)+k=E-k-2$$

由于新树比老树的树叶更少，由归纳假设我们知道：

$$E-k-2=(I-k)+2(q-1)$$

重新安排这个等式，则给出预想的结果。（**证明结束**）

5. 关于 binary_search_2 的成功查找的分析

如我们所知，在 binary_search_2 的比较树中，到树叶的距离是 $\lg(n+1)$。树叶的数目是 $n+1$，因此外部路径长度大约是 $(n+1)\lg(n+1)$。定理 7.3 表明内部路径长度大约是 $(n+1)\lg$

$(n+1)-2n$。为了得到成功查找所进行的平均比较次数，我们必须首先除以 n(非树叶数)，然后再加 1 乘 2，因为在每个内部顶点会进行两次比较。最后，我们再减去 1，因为在找到目标的顶点上只进行一次比较。因此结果是：

binary_search_2 在 n 个元素的表上的成功查找大约要进行

$$\frac{2(n+1)}{n}\lg(n+1)-3$$

次键比较。

7.4.3 方法的比较

* * * 简化后的次数

注意两个版本的二分查找的公式的相似性和差别。回忆一下，首先我们在计算中已做了一些近似处理，因此我们的公式只是近似的。对较大的 n 值，$\lg n$ 和 $\lg(n+1)$之间的差别是无关紧要的，而且$(n+1)/n$ 也非常接近 1，因此我们如下简化我们的结果：

	成功的查找	不成功的查找
binary_search_1	$\lg n+1$	$\lg n+1$
binary_search_2	$2\lg n-3$	$2\lg n$

在所有的 4 种情况中，除了很小的常数项，它们的时间都与 $\lg n$ 成正比，并且在所有情况中，$\lg n$ 的系数是循环中的比较的次数。binary_search_2 中的循环能早一些终止的事实对成功查找的性能改进的贡献少得令人失望，它完全没有降低 $\lg n$ 的系数，但仅将常数项从 +1 降为 -3。

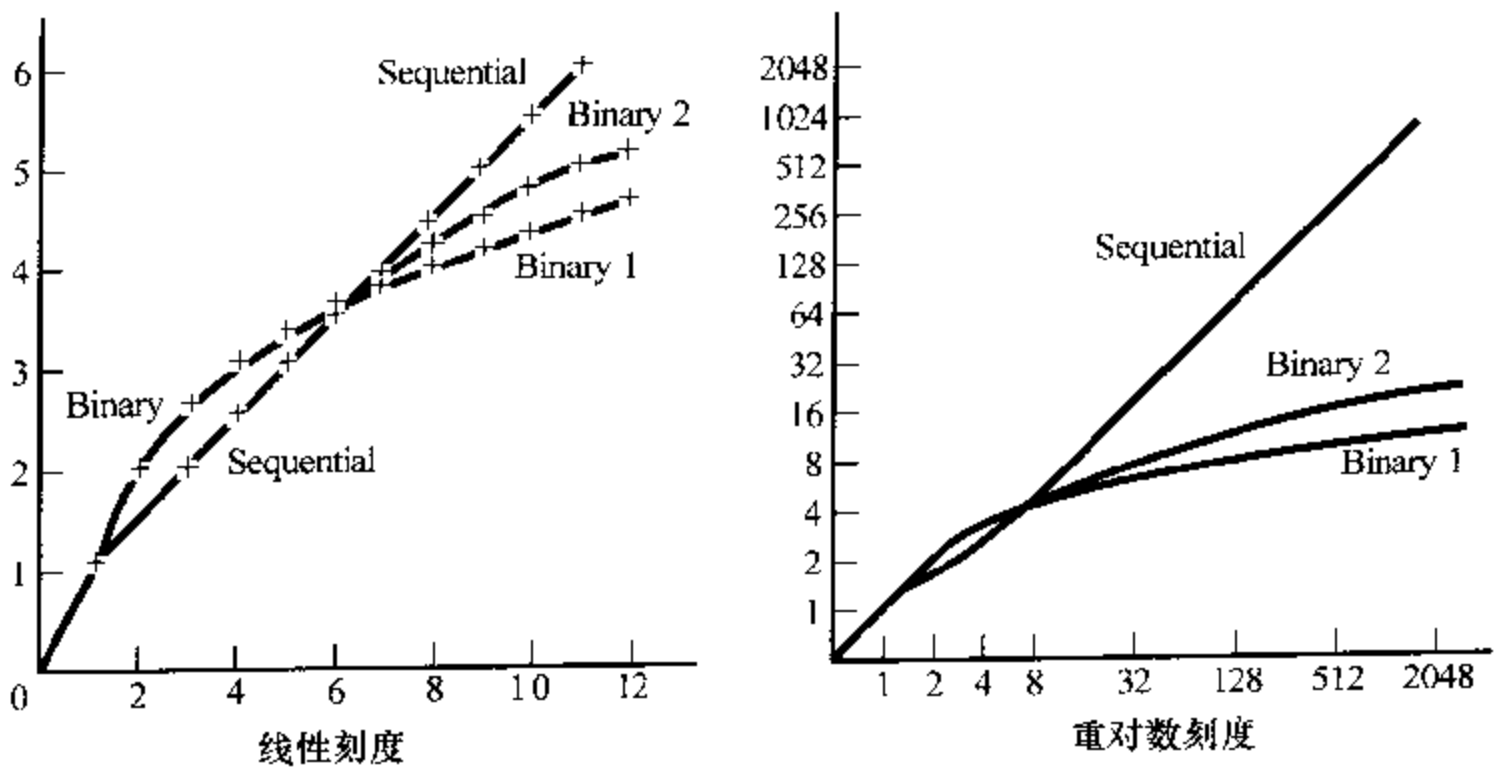
对比较树进行片刻的检查会说明其原因。超过一半的顶点出现在底层，因此它们的循环不能早一些终止。剩下的顶点中又有超过一半只能早一轮终止。因此对较大的 n，在树的相对高的上半部分中的顶点数与底层的顶点数相比是微不足道的。只有对这微不足道的一部分顶点，binary_search_2 能达到比 binary_search_1 更好的结果，但它付出的代价却是对包括成功的和不成功的所有查找，近乎进行两倍的比较。

binary_search_1 有更小的 $\lg n$ 的系数，当 n 足够大时，它会做较少的比较。但 binary_search_2 有较小的常数项，当 n 较小时，它能进行较少的比较。但是对这样小的 n 值，建立二分查找的系统开销和额外的程序设计强度使得对它的使用可能比顺序查找代价更大。因此我们得出的结论是 binary_search_2 可能并不值得努力，因为对大型问题，binary_search_1 更加出色，而对小的问题，sequential_search 更好，这与我们直觉得到的结论完全相反。然而为公平起见，采用某些计算机和优化的编译器，binary_search_2 中需要的两次比较不会占用 binary_search_1 中一次比较的两倍时间，因此在这种情况下 binary_search_2 可能证明是更好的选择。

进行算法分析的目的是帮助我们判断在适当的情况下哪一个算法更好。不管前述的限制条件，我们现在能够做出这样的决定，并且有了信息可用，而这些信息否则就不明显。

* * * 对数图

sequential_search,binary_search_1 和 binary_search_2 在平均成功情况下所进行的键比较次数显示在图 7.7 中,图中所示的数字来自于对这些函数的测试运行,它们不是近似值。图 7.7 中的第 1 个图用较小的 n 值比较 3 个函数,n 为表中的元素数。在第 2 个图中,我们应用重对数图(log-log graph)对更大范围的数进行比较,其中轴上的每个单元表示相应坐标的两倍。在第 3 个图中我们希望比较两个版本的二分查找,这里采用半对数图(semilog graph)较为合适,因此垂直的轴保持线性单元,而水平的轴是对数的。



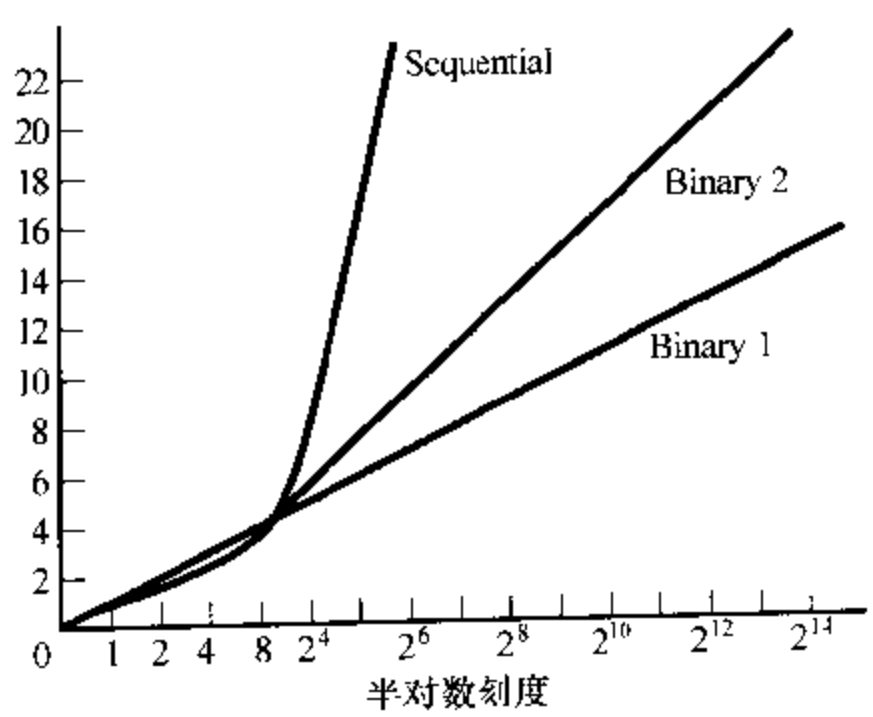


图 7.7 平均成功查找的比较次数

7.4.4 一般关系

* * * 假设

在结束这一节之前,我们使用定理 7.3 来获得成功查找与不成功查找的平均键比较次数之间的一个关系。这一关系对任何查找方法都成立,只要它的比较树能画成像 binary_search_2 的比较树那样。即,我们假设比较树的树叶对应于不成功的查找,而内部顶点对应

于成功的查找，并且除了找到目标的顶点外，对每个内部顶点都要进行两次比较。如果 I 和 E 分别是此树的内部和外部路径长度，且 n 是表中的元素数，因此 n 也是树中的内部顶点数，那么，与 binary_search_2 中的分析一样，在成功的查找中，平均比较次数为：

$$S=2\left(\frac{1}{n}+1\right)-1=\frac{2I}{n}+1$$

并且不成功查找的平均次数为 $U=2E/(n+1)$。由定理 7.3，$E=I+2n$。结合这些表达式，可以推断：

定理 7.4 特定条件下，成功和不成功查找中键比较的平均次数可以用下式表示：

$$S=\left(1+\frac{1}{n}\right)U-3$$

换言之，成功查找中键比较的平均次数几乎恰好与不成功查找一样。因此如果查找是采用键比较的方式，则已知某个元素在表中对找到它的帮助并不大。

练习 7.4

E1. 分别画出当(1)$n=5$，(2)$n=7$，(3)$n=8$，(4)$n=13$ 时(i)binary_search_1 和(ii) binary_search_2 的比较树。计算这些树中每一个的外部和内部路径长度，并验证定理 7.3 的结论成立。

E2. 顺序查找比二分查找的系统开销更少，因此对小的 n 值可能运行更快。找出 sequential_search 和 binary_search_1 进行同样次数的键比较时的平衡点。按照公式计算成功查找中所进行的平均比较次数。

E3. 假设在一个数组中有一个以字母顺序排列的 10 000 个名字的表，而且必须经常查找各种名字。有结果表明 20%的名字占据了 80%的检索。取代每次在 10 000 个名字中进行二分查找，考虑将此表分成两个表的可能性：一个包含 2 000 个名字的高频表和一个包含剩余 8 000 个名字的低频表。为查找一个名字，首先在高频表中使用二分查找，在 80%的情况下，不必进入第二阶段，即不必在低频表上使用二分查找。这种方案是否值得努力？通过比较 binary_search_1 在新的方案中的平均成功查找比较次数和在 10 000 个名字的单个表中二分查找的平均成功查找比较次数，证明你的答案。

E4. 使用 n 上的数学归纳法证明：设 $n>0$，binary_search_1 在 n 个元素的表上的比较树中，所有的树叶在$\lfloor \lg 2n \rfloor$和$\lceil \lg 2n \rceil$层上。（因此，如果 n 是 2 的幂(所以 $\lg 2n$ 是一个整数)，则所有的树叶在一层，否则，它们都在两个邻接层上。）

E5. 修改二分查找，使得它每次实质上不是将表分成两半，而是分成长度约 1/3 和 2/3 大小的两段，则它的平均比较次数近似为何？

程序设计项目 7.4

P1. (1)编写一个与 binary_search_2 类似的“三重”查找函数，它检查整个表中 1/3 处的键，且若目标键更大时，再检查表的 2/3 处的键，因此无论如何在每一遍都将表的长度减少

到 1/3。(2)将你的函数作为一个附加的选项,包含在 7.2 节项目 P1 的测试程序中,并与其他的方法比较性能。

P2. (1)编写一个程序进行"混合的"查找,对较大的表使用 binary_search_1,当查找减少到足够小的子表时转换到顺序查找。(由于不同的系统开销,最佳的转换点未必与练习 E2 中的答案一样。)(2)将此函数作为一个附加的选项,包含在 7.2 节项目 P1 的测试程序中,并与其他的方法比较性能。

7.5 下限

我们知道对有序的顺序表,二分查找比顺序查找更快。人们很自然地就会询问是否能找到另一种比二分查找更快的方法。

1. 润饰程序

一种方法是尝试润饰和提炼我们的程序以使它们运行得更快。我们可以巧妙地将每次重复进行的工作减到很少并因此加速算法。有一种方法称为斐波纳契查找(Fibonacci search),它设法用某些减法去代替二分查找的循环中的除法(不需要辅助的表),这在某些计算机上会加速此函数。

*** 基本算法和细小的变化

巧妙地调整程序的性能可以将它的运行时间减至一半,或许能减去更多,但如果潜在的算法不做改变的话,很快会达到极限。二分查找比顺序查找快得多的原因不在于其循环内有更少的步骤(实际上有更多)或者它的代码经过了优化,而是循环重复的次数更少,只做 $\lg n$ 次而不是 n 次。当 n 增长时,$\lg n$ 的值比 n 的值增长更加缓慢。

在比较潜在方法的上下文中,binary_search_1 与 binary_search_2 之间的差别同二分查找和顺序查找之间的差别相比就显得无关紧要。对大型的表,binary_search_2 会要求近两倍 binary_search_1 的时间,但同 $2\lg n$ 次比较与顺序查找有时候需要的 n 次比较之间的差别相比,$2\lg n$ 与 $\lg n$ 之间的差别就微不足道了。

2. 任意的查找算法

现在让我们来问问是否可能存在任何查找算法,它在最坏和平均情况下,都能够比二分查找用显著少的键比较次数来找到它的目标。倘若停留在仅依靠键比较去判断在有序表中何处进行查找这类算法,我们将看到这个答案是否定的。

*** 通用算法和比较树

我们从通过键比较查找有序表的任意算法开始,并用为 binary_search_1 画比较树的同样方式想像地画出它的比较树,即树的每个内部顶点将对应于一次键比较,而每个树叶对应于一次可能的最终结果。(如果此算法表达成像 binary_search_2 的那些三路比较,则将每个内部顶点扩充成两个,就如图 7.4 中为一个顶点所显示的。)树叶对应的可能的结果不仅包括目标的成功发现,而且也包括算法可以判别的不同种类的失败。长度为 n 的表的二分查找产生 $k=2n+1$ 种结果,它由 n 个成功的结果和 $n+1$ 个不同种类的失败(小于最小的键,在每对键之间或大于最大的键)组成。另一方面,我们的顺序查找函数仅产生 $k=n+1$ 种可

能的结果，因为它仅判别一种失败。

* * * 高度和外部路径长度

同所有的比较键的查找算法一样，树的高度将等于算法在其最坏情况下所进行的比较次数，而(由于所有的结果对应于树叶)树的外部路径长度除以可能的结果数将等于算法所进行的平均比较次数。因此我们希望根据树叶的数目 k 得到高度和外部路径长度的下限。

3. 对2-树的观察

我们所需要的关于2-树的结果如下：

引理7.5 假设 T 是具有 k 个树叶的2-树，则 T 的高度 h 满足 $h \geqslant \lceil \lg k \rceil$，$T$ 的外部路径长度 $E(T)$ 满足 $E(T) \geqslant k \lg k$。当 T 的所有树叶在同一层或在两个相邻层上时，h 和 $E(T)$ 出现最小值。

证明 我们通过在后一个句子中建立断言开始证明。假设 T 的某些树叶在 r 层，某些树叶在 s 层，并且 $r>s+1$。现在从 r 层取两个树叶，它们都是同一顶点 v 的孩子，我们将这两个树叶从 v 顶点分开，并将它们附到 s 层某个(以前是)树叶上作为它的孩子。则我们将 T 变成一棵新的2-树 T'，T' 仍然有 k 个树叶，其高度必定不超过 T，并且因为 $r>s+1$，则 T' 的外部路径长度满足 $E(T')=E(T)-2r+(r-1)-s+2(s+1)=E(T)-r+s+1<E(T)$。此表达式中的各项如下获得：由于从层 r 移去两个树叶，则将 $E(T)$ 减少 $2r$；由于顶点 v 变成了树叶，则将 $E(T)$ 加上 $r-1$；由于 s 层的顶点不再是一个树叶，则将 $E(T)$ 减去 s；由于以前在 r 层的两个树叶现在在 $s+1$ 层，则将项 $2(s+1)$ 加到 $E(T)$ 上。此过程在图7.8中说明。

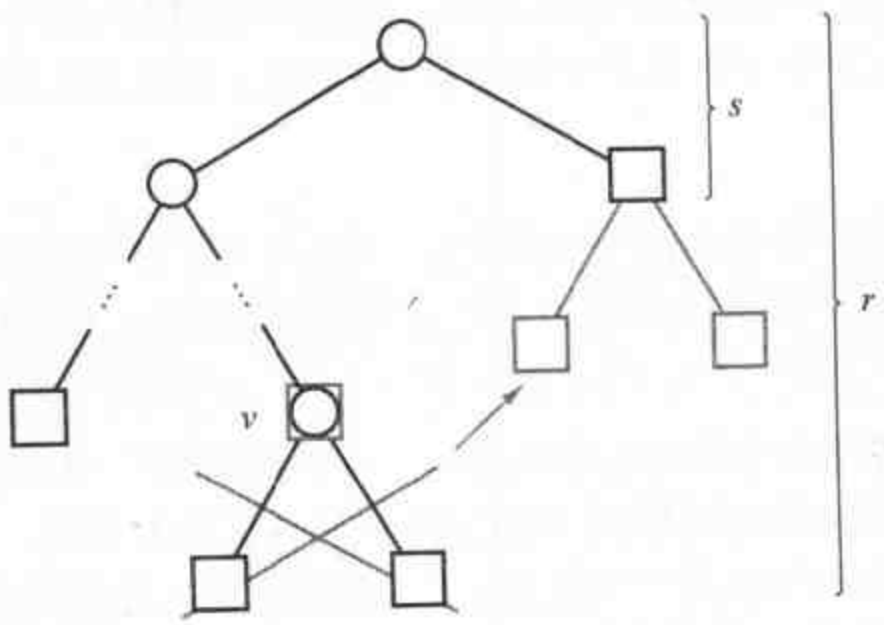


图7.8 在2-树中将树叶上移

我们继续以此方式将树叶移到树的更高层，每次减少外部路径长度并可能减少树的高度，直到最后所有的树叶都位于同一层或相邻层上，并且树的高度和外部路径长度将是所有 k 个树叶的2-树中最少的。

* * * $h \geqslant \lceil \lg k \rceil$ 的证明

为证明引理7.5中剩余的断言，我们从现在起，假设 T 在所有 k 个树叶的2-树中具有最小的高度和路径长度，因此 T 的所有的树叶都出现在 h 层和(可能) $h-1$ 层，这里 h 是 T

的高度。由引理 7.2，h 层的顶点数（它们必定是树叶）最多为 2^h。如果所有的树叶都在 h 层，则 $k \leqslant 2^h$。如果某些树叶在 $h-1$ 层，则这一些树叶中的每一个（因为它没有孩子）都会将 h 层上的树叶减少 2 个，因此界限 $k \leqslant 2^h$ 仍然成立。我们应用对数得到 $h \geqslant \lg k$，并且由于高度总是一个整数，因此我们提升到上限 $h \geqslant \lceil \lg k \rceil$。

* * * $E(T) \geqslant k\lg k$ 的证明

对于外部路径长度的界限条件，假设 x 表示 T 在 $h-1$ 层上的树叶数，因此在 h 层有 $k-x$ 个树叶。这些顶点正好是 $h-1$ 层中 $\frac{1}{2}(k-x)$ 个顶点的孩子，它们与 x 个树叶组成了 $h-1$ 层上所有的顶点。因此由引理 7.2，$\frac{1}{2}(k-x)+x \leqslant 2^{h-1}$，它转换成 $x \leqslant 2^h-k$。我们现在有：

$$\begin{aligned} E(T) &= (h-1)x + h(k-x) \\ &= kh - x \\ &\geqslant kh - (2^h - k) \\ &= k(h+1) - 2^h \end{aligned}$$

从高度的界限，我们已知 $2^{h-1} < k \leqslant 2^h$。如果设置 $h = \lg k + \varepsilon$，则 ε 满足 $0 \leqslant \varepsilon < 1$，如果将 ε 替换成 $E(T)$ 的界限，我们得到 $E(T) \geqslant k(\lg k + 1 + \varepsilon + 2^{\varepsilon})$。它表明，对 $0 \leqslant \varepsilon < 1$，$1+\varepsilon-2^{\varepsilon}$ 的量是在 0～0.0861 之间。因此最小路径长度非常接近 $k\lg k$，并且我们将说明，在任何情况下，它至少是 $k\lg k$。到此，完成引理 7.5 的证明。（**证明结束**）

4. 查找的下限

最后，我们回过头来研究任意的查找算法。在它的比较树中，不一定所有的树叶都在两个相邻层上，但即使是这样，引理 7.5 中的界限仍然成立。因此我们可以将这些界限转换成如下的比较语言。

定理 7.6 假设算法使用键比较来查找表中的目标，如果有 k 种可能的结果，则此算法在其最坏情况下必须至少进行 $\lceil \lg k \rceil$ 次键比较，而在平均情况下至少进行 $\lg k$ 次键比较。

注意最坏情况的界限和平均情况的界限只有非常小的差别。另外由定理 7.4，对许多算法来说，在决定前面的定理中的界限时，查找成功与否并无太大关系。当我们将定理 7.6 用于诸如二分查找这样的算法时，由于长度为 n 的有序表上的二分查找有 n 个成功和 $n+1$ 个不成功的结果，则我们得到最坏情况的键比较的界限为 $\lceil \lg(2n+1) \rceil \geqslant \lceil \lg(2n) \rceil = \lceil \lg n \rceil + 1$，平均情况的键比较的界限为 $\lg n + 1$。当我们将这些数字同 binary_search_1 的分析中得到的数字进行比较时，我们得到：

推论 7.7 binary_search_1 在所有的通过键比较来查找有序表的算法中是最佳的。在平均和最坏情况下，binary_search_1 都达到了最佳的界限。

理解推论 7.7 为何正确的一种非正式的方法是从任一查找算法开始，并想像地画出它对长度为 n 的表的比较树。由于此算法是任意的，我们实际上并不能画出这棵树，但它却存

在，因此我们能在想像中利用它工作。如果这棵树碰巧有一些树叶比树中的其他树叶高至少两层，则可以通过从最低层删除一对兄弟树叶并将它重新接在至少高两层以上的先前的树叶上作为它的孩子，从而修改这棵树。（这一过程在图 7.8 中说明。）这样做会缩短树的路径长度。现在想像我们甚至能够修改任意的算法，使得它的比较树变成修改后的树，因为键的比较次数与树的路径长度密切相关，所以这样做会改进算法的性能。现在只要比较树中还有任何树叶比其他的树叶高至少两层，那么就继续用同样的方式优化算法。这样做会使算法越来越好，直到最终因为其比较树中的所有树叶是在一层或两个相邻层上，不能再以此方式对它做进一步的优化。但是 binary_search_1 已有一棵像这样的树。换言之，通过从任一查找算法开始并尽可能地优化它，可能将它的性能提高到 binary_search_1 的水平，因此 binary_search_1 就是我们目前得到的最好的算法。

5. 其他查找方法

定理 7.6 中的界限并不意味没有其他的算法比二分查找更快，只是意味着对基于键比较的那些算法是如此。作为一个简单的例子，假设键是 $0 \sim n-1$ 之间的整数，如果我们知道目标键 x 是此范围内的一个整数，则我们决不会执行查找算法来定位它的元素，我们只是将这些元素存储在长度为 n 的数组中，而且查看下标 x 即可以找到想要的元素。

* * * 插值查找

可以将这种思想进行扩展，得到另一种称为插值查找（interpolation search）的方法。假设键为数字或信息，例如单词，它们都能够轻易地编码成数字。此方法也假设表中的键是均匀分布的，即键在某特定范围内的概率等于它在同样长度的任何其他范围内的概率。为找到目标键 target，插值查找按照与表中第一个和最后一个元素与数字 target 相关的量级，估计 target 会在表中哪个位置并在那里进行查看，然后它按照 target 小于或大于要检查的键减少表的长度。可以表明，平均情况下，对均匀分布的键，插值查找要进行大约 $\lg \lg n$ 次键比较，对较大的 n，这个数比二分查找所要求的稍微小些。例如，当 $n=1\ 000\ 000$ 时，binary_search_1 需要约 $\lg 10^6+1 \approx 21$ 次比较，而插值查找只需要约 $\lg \lg 10^6 \approx 4.32$ 次比较。

插值查找是人们通常用来在一本书中查找特定一页的方法。如果你猜测一本书约有 300 页厚，并且希望找出第 345 页，通常你首先会翻到书的全部厚度的 2/3 处并从那里开始查找。如果你希望找出第 10 页，则你会从接近开头处开始。或者从结尾开始找第 487 页。

最后，我们要重复的是，即使对通过比较进行的查找，要求所有的键具有同等可能性的假设完全不正确。如果有一或两个键比其他的键更可能出现，则即便是顺序查找，如果它首先查看那些键的话，也可能比其他方法更快。更普遍地，查找或信息检索具有根本的重要性，已投入大量的工作，致力于研究它们的方法。在后继章节中，我们将再三地返回到这些问题。

练习 7.5

E1. 假设一个查找算法像 binary_search_2 那样进行三路比较，它的比较树中每个内部顶点对应于一次成功的查找，而每个树叶对应于一次不成功的查找。

(1)使用引理 7.5 得到如同定理 7.6 那样的一个定理，给出这样的算法对不成功的查找

在最坏和平均情况下性能的下限。

(2)使用定理 7.4 得到成功查找的类似的结果。

(3)将你得到的边界与 binary_search_2 的分析进行比较。

程序设计项目 7.5

P1. (1)编写程序实现插值查找并验证其正确性(特别是终止),参见本章后面的参考书目以获得建议和程序分析。(2)将你的函数作为另一个选项加入 7.2 节项目 P1 的测试程序中,并将它的性能与其他方法进行比较。

7.6 渐近

7.6.1 引言

现在已到了从对查找算法所进行的分析中提取出重要的一般结论的时候了。随着我们的进展,我们能更清楚地了解算法分析的哪些方面很重要,而哪些方面确实能予以忽略。例如,如果一个程序的某个部分仅在任何循环或递归外执行一次,则它所用的时间量同循环或递归内所用的时间量相比可以忽略不计。我们已发现,虽然二分查找比顺序查找更难以进行程序设计和分析,并且即使当它应用于很短的表时运行得更慢,但对较长的表,它会运行得远比顺序查找快。

* * * 设计小型问题的算法

适用于小型问题的有效方法的设计是一个重要的研究主题,因为大型问题在其执行时可能需要多次完成同样的或类似的小任务。然而就像我们所发现的,复杂方法的大量的系统开销使它次于更简单的方法。对三、四个元素长的表,顺序查找确实优于二分查找。为提高小型问题的算法的效率,程序设计员有必要将注意力放在计算机系统和程序设计语言的细节特性上,然而在这方面全面的有用的观察资料却很少。

* * * 大型问题的方法选择

用于大型问题的有效算法的设计则完全是另外一回事,而这也正是我们现在所关心的。例如在研究查找方法中,我们已看到当表的长度增长时,系统开销变得相对不重要。这是区分高效算法和由于太慢而不实用的算法的基本思想。

* * * 渐近

作为本节标题的"渐近"(asymptotics)这个词意味着当 n 无限地变得越来越大时,对参数为 n 的函数的研究。典型地,我们研究函数 $f(n)$,随着参数 n 的增长,它给出了算法在解决长度为 n 的问题中的工作量。例如,在比较 n 个元素的表的查找算法时,我们看到键比较次数 $f(n)$ 正确地反映了大型问题的运行时间,因为所有其他的操作(如增量或比较下标)都与键比较前后紧接地进行,因此它普遍成立。

* * * 基本动作

* * * 程序变化

实际上,这种基本动作的频率比包括内务管理在内的所有操作的合计总数更加重要。

包括内务管理在内的总数太依赖于程序设计语言的选择和程序设计员的特殊风格，这种依赖程度简直要遮掩了通用的方法。内务管理细节或程序设计技巧方面的变化能轻易地使程序运行时间增至3倍，但这样的变化不会区别计算是否可行。如果我们再等等或者再多投资点钱，我们能够得到3倍快的计算机，因此对占用3倍时间的程序不会感到不便。

＊＊＊方法选择

另一方面，在基本方法中所做的一点修改会给求解问题所需的资源带来重大的差别。如果一个算法所做的基本操作数与输入长度 n 成比例，则无论怎样完成内务管理，将 n 翻倍大约会使运行时间翻倍。如果基本操作数与 $\lg n$ 成比例，则将 n 翻倍几乎完全不会改变运行时间。如果基本操作数与 n^2 成比例，则运行时间将会是4倍，而计算仍然是可行的，但这么长的时间会令人不舒服。现在假设基本操作数与 2^n 成比例，即为 n 的指数函数，在此情况下，将 n 翻倍会将程序必须做的基本操作数自乘。花费1秒钟的计算可能包含百万个(10^6)基本操作，因此将输入长度翻倍会要求 10^{12} 个基本操作，将运行时间从1秒增加到 $11\frac{1}{2}$ 天。再次将输入翻倍会将基本操作数增加到 10^{24}，时间增长到300亿年。函数 2^n 当 n 增长时确实增长迅速。

＊＊＊普遍性

我们希望系统地阐述适用于分析多种算法的普遍原理，用一个符号正确地反映计算时间随长度增长的方式，同时忽略对全局影响很小的不必要的细节。我们想集中于算法中的一个或两个基本操作，而不怎么关心伴随它们的所有内务管理操作。如果输入长度为 n 的算法做 $f(n)$ 次基本操作，则它的全部运行时间将最多是 $cf(n)$，这里 c 是一个常数，依赖于对算法进行程序设计的方式和所用的计算机，但当输入的长度 n 增长时，c 不依赖于 n。

我们现在的目标是获得一个简洁的、易于理解的符号，它将告诉我们当 n 增长时函数 $f(n)$ 增长多快，这个符号向我们提供有关算法所做工作量的有用的信息。

7.6.2 数量级

1. 定义

我们的思想是将自己的函数 $f(n)$ 与某个众所周知的已知行为的函数 $g(n)$ 进行比较。实际上，常常选择下列 $g(n)$ 函数用于比较 $f(n)$：

- $g(n)=1$ 常函数
- $g(n)=\log n$ 对数函数
- $g(n)=n$ 线性函数
- $g(n)=n^2$ 二次函数
- $g(n)=n^3$ 三次函数
- $g(n)=2^n$ 指数函数

为了用 $g(n)$ 比较 $f(n)$，我们取 $f(n)/g(n)$ 的商并计算 n 无限增长时这个商的极限。依赖于结果，有下列情况之一：

定义 如果 $\lim\limits_{n\to\infty}\dfrac{f(n)}{g(n)}=0$，则：$f(n)$ 有比 $g(n)$ 严格小的数量级。

如果$\lim_{n\to\infty}\frac{f(n)}{g(n)}$是非零的有限值，则：$f(n)$与$g(n)$有相同的数量级。

如果$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$，则：$f(n)$有比$g(n)$严格高的数量级。

当上下文的意义明确时，数量级(order of magnitude)这个术语经常缩写为阶(order)。

注意当$f(n)$和$g(n)$有同样的数量级时，第二种情况包含除0和∞外极限的所有值。这样，用任何非零的常数因子改变一个算法的运行时间将不会影响其数量级。

2. 假设

在此定义中，并且一直贯穿本书，我们做出两个假设：

- 假设对所有足够大的n，有$f(n)>0$且$g(n)>0$。
- 假设$\lim_{n\to\infty}\frac{f(n)}{g(n)}$存在。

假设$f(n)$和$g(n)$为严格正(对大的n)是为了避免被0除的可能性以及避免对极限正负担心。由于算法的操作次数和计时始终是正的，因此实际上并无限制。

极限存在的假设避免了商是像$x\sin x$这样的函数的可能性，图7.9中显示了$x\sin x$的图的一部分，此函数无限次地采用每个可能的值，无论要求多大的x，它对甚至更大的x的值仍无限次采用每个可能的值。因此当x趋于无限时，无法认为$x\sin x$逼近任何固定的极限。

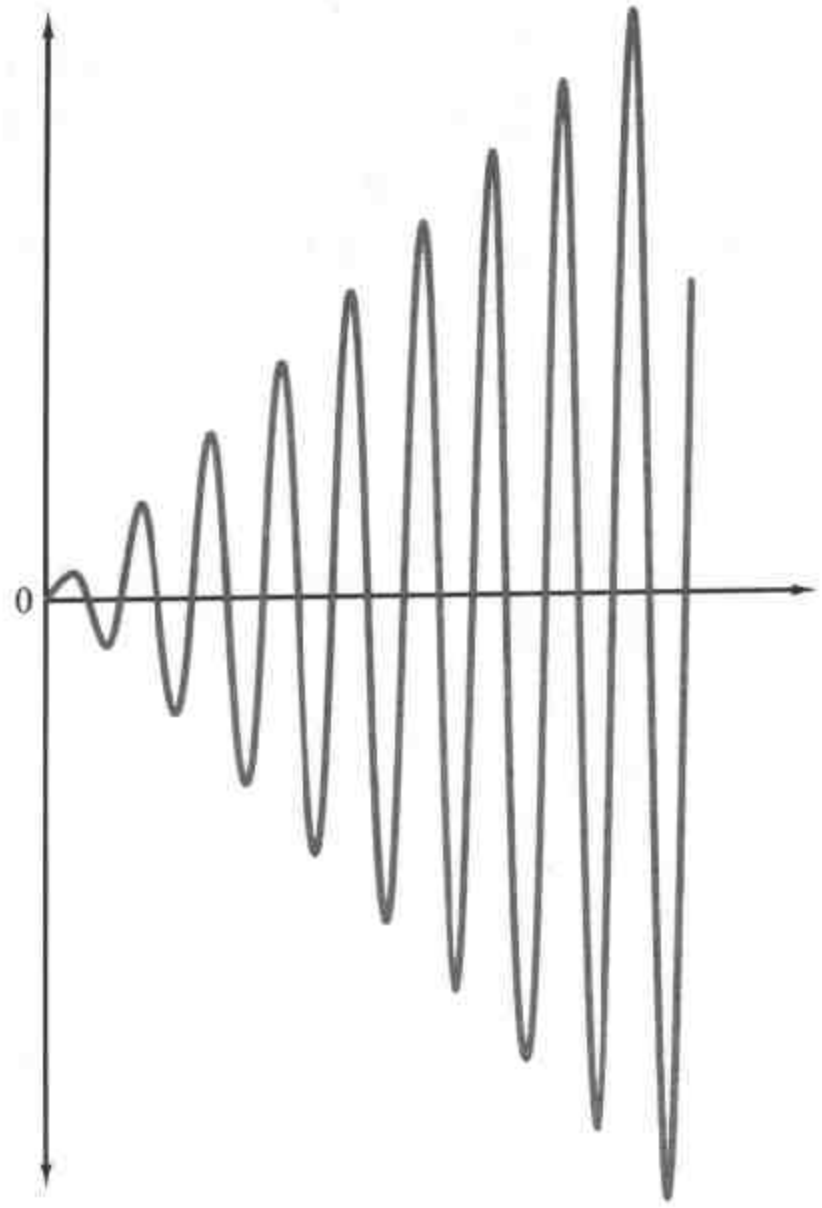


图7.9 $x\sin x$的图

注意我们遵循下面的约定，即极限越来越大、没有边界的函数确实存在并且是无限的。（某些数学家认为这样的极限并不存在。）下面根据有限极限对无限极限加以定义：

定义 $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\infty$ 与 $\lim_{n\to\infty}\frac{g(n)}{f(n)}=0$ 具有同样的意义。

3. 多项式

作为第一个例子，我们来看多项式 $f(n)=3n^2-100n-25$。首先，注意对小的 n 值，$f(n)$ 是负的，例如 $f(1)=3-100-25=-122$。这是由于对小的 n，$3n^2<100n$。但是随着 n 的增长——实际上在 $n\geqslant 34$ 后的任何时候——$3n^2$ 高于 $100n$ 时，实际上对所有的 $n\geqslant 34$ 都有 $f(n)>0$。

假设 $g(n)=n^3$，则：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{3n^2-100n-25}{n^3}=\lim_{n\to\infty}\left(\frac{3}{n}-\frac{100}{n^2}-\frac{25}{n^3}\right)=0$$

由于上式中每一项当 $n\to\infty$ 时都趋于 0，因此 $3n^2-100n-25$ 有比 n^3 严格小的阶。

另一方面，对 $g(n)=n$，我们有：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{3n^2-100n-25}{n}=\lim_{n\to\infty}\left(3n-100-\frac{25}{n}\right)=\infty$$

由于当 $n\to\infty$ 时，上式中第一项趋于 ∞，而第二项不变，且第三项趋于 0，因此 $3n^2-100n-25$ 有比 n 严格高的阶。

如果我们选择 $g(n)=n^2$，我们得到与 $f(n)$ 同样的阶，因为：

$$\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{3n^2-100n-25}{n^2}=\lim_{n\to\infty}\left(3-\frac{100}{n}-\frac{25}{n^2}\right)=3$$

很容易概括这个例子，得到：

* * * 多项式

若 $f(n)$ 是 n 的 r 次任意多项式，则 $f(n)$ 有与 n^r 同样的数量级。

我们也能看到：

* * * n 的幂

若 $r<s$，则 n^r 有比 n^s 严格小的数量级。

4. 对数和 L′Hôpital 规则

对数形成了算法研究中频繁出现的第二类函数。在二分查找的分析中已经使用了对数，而且 n 的对数比 n 本身的增长更缓慢，我们现在要对这种观察进行归纳，但我们首先指出下面的事实：

* * * 底数的变化

对数的数量级不依赖于对数的底数。

为理解其正确性，设 $\log_a n$ 和 $\log_b n$ 是两个不同底数的对数，其中 $a>1$ 和 $b>1$，如 A.2.6 节，$\log_b n=(\log_b a)(\log_a n)$。因此，

$$\lim_{n\to\infty}\frac{\log_b n}{\log_a n}=\lim_{n\to\infty}\frac{(\log_b a)(\log_a n)}{\log_a n}=\log_b a$$

这是一个非零常数，因此 $\log_b n$ 和 $\log_a n$ 具有同样的数量级，我们将对此给出说明。

由于对数的底数不影响数量级，因此我们通常在任何数量级表达式中应写成 log 而非 lg 或 ln。

接下来，让我们比较 $\log n$ 与 n 的幂如 n^r 的数量级，这里 $r>0$ 是任何正实数。当我们取商时，我们看到当 $n\to\infty$ 时，$\log n\to\infty$ 且 $n^r\to\infty$。这个商的极限被称为不确定的，因为不可能不用更多的信息就能确定这个极限。可以从微积分中借用一个重要工具来帮助解决这个问题，此工具称为 L'Hôpital 规则。

定理 7.8 L'Hôpital 规则 假设：

- 对所有足够大的 x，$f(x)$ 和 $g(x)$ 是可微函数，其导数分别是 $f'(x)$ 和 $g'(x)$。
- $\lim_{x\to\infty}f(x)=\infty$ 并且 $\lim_{x\to\infty}g(x)=\infty$。
- $\lim_{x\to\infty}\frac{f'(x)}{g'(x)}$ 存在。

则 $\lim_{x\to\infty}\frac{f(x)}{g(x)}$ 存在并且 $\lim_{x\to\infty}\frac{f(x)}{g(x)}=\lim_{x\to\infty}\frac{f'(x)}{g'(x)}$。

当我们将 L'Hôpital 规则应用于 $f(x)=\ln x$ 和 $g(x)=x^r$，$r>0$ 时，我们有 $f'(x)=1/x$ 并且 $g'(x)=r\,x^{r-1}$，由于 $r>0$，因此：

$$\lim_{x\to\infty}\frac{\ln x}{x^r}=\lim_{x\to\infty}\frac{1/x}{rx^{r-1}}=\lim_{x\to\infty}\frac{1}{rx^r}=0$$

因为对数的底数并无关系，我们有：

* * * 对数

$\log n$ 有比 n 的任何正数次幂 $n^r(r>0)$ 严格小的数量级。

5. 指数函数

我们能再次应用 L'Hôpital 规则验证指数函数有比 n 的幂严格高的数量级。特别地，设 $f(x)=a^x$，其中 $a>1$ 是一个实数，并设 $g(x)=x^r$，其中 r 是一个正整数。由于当 $x\to\infty$ 时，$f(x)\to\infty$ 且 $g(x)\to\infty$，我们计算导数：

$$f'(x)=\frac{d}{dx}a^x=\frac{d}{dx}(e^{\ln a})^x=\frac{d}{dx}e^{(\ln a)x}=(\ln a)e^{(\ln a)x}=(\ln a)a^x \text{ 且 } g'(x)=rx^{r-1}$$

令人遗憾的是，当 $x\to\infty$ 时，$f'(x)\to\infty$ 且（如果 $r>1$）$g'(x)\to\infty$，因此 L'Hôpital 规则不能直接给出解答。但是我们能将 L'Hôpital 规则应用于 $f'(x)$ 和 $g'(x)$。再次，$f'(x)/g'(x)$ 的商可能无法确定，但我们能一直做到第 r 次导数，此时我们得到：

$$f^{(r)}(x)=(\ln a)^r a^x \text{ 并且 } g^{(r)}(x)=r!$$

最终，这个商不再是无法确定的：$f^{(r)}(x)\to\infty$ 且 $g^{(r)}(x)$ 是常数 $r!$。因此：

$$\lim_{x\to\infty}\frac{f(x)}{g(x)}=\lim_{x\to\infty}\frac{f'(x)}{g'(x)}=\cdots=\lim_{x\to\infty}\frac{f^{(r)}(x)}{g^{(r)}(x)}=\lim_{x\to\infty}\frac{(\ln a)^r a^x}{r!}=\infty$$

因此：

* * * 指数

对任意实数 $a>1$ 和任意正整数 r，任何指数函数 a^n 有比 n 的任何次幂 n^r 严格高的数量级。

最后，让我们比较两个不同底数的指数函数 a^n 和 b^n。我们能再次试用 L'Hôpital 规则，但我们现在能发现当 $x\to\infty$ 时，他们的导数总是趋于无限。幸运的是，我们能直接判定极限。我们假设 $0\leqslant a<b$，

$$\lim_{x\to\infty}\frac{a^n}{b^n}=\lim_{x\to\infty}(\frac{a}{b})^n=0$$

由于 $a<b$，因此：

* * * 两个指数

若 $0\leqslant a<b$，则 a^n 有比 b^n 严格小的数量级。

6. 通用阶

我们能继续比较其他函数的数量级，但幸运的是，对我们需要的大多数算法分析，只需要一个很短的函数表。我们已了解了 6 个函数，它们是 1（常数）、$\log n$（对数）、n（线性）、n^2（二次）、n^3（三次）和 2^n（指数）。从我们所完成的工作中，我们能得到这 6 个函数的数量级排列严格递增这一结论。

在 8.5 节，我们将看到另一个函数也很重要，这个函数是 $g(n)=n\log n$。在数量级中，这个函数适合表中的哪个位置呢？为回答这一问题，我们要用到下面的简单事实：

* * * 函数的积

若 $h(n)$ 是任意函数，对所有足够大的 n 有 $h(n)>0$，则 $f(n)h(n)$ 的阶（数量级）与 $g(n)h(n)$ 的阶的关系同 $f(n)$ 的阶与 $g(n)$ 的阶的关系具有同样的方式（小于、等于或者大于）。

对它的证明只是观察到：

$$\lim_{n\to\infty}\frac{f(x)h(x)}{g(x)h(x)}=\lim_{n\to\infty}\frac{f(x)}{g(x)}$$

首先，让我们比较 $n\log n$ 和 n 的阶。我们设 $h(n)=n$，则可以看到这个比较与 $\log n$ 和 1 之间的比较一样，因此我们得出结论：$n\log n$ 有比 n 严格高的阶。但是如果取任何 $\epsilon>0$，则 $\log n$ 有比 n^ϵ 严格低的阶。因此（再次用 $h(n)=n$）$n\log n$ 有比 $n^{\epsilon+1}$ 严格低的阶。

图 7.10 说明这 7 个函数（常数为 1 且对数的底数为 2）如何随 n 增长，在图 7.11 中给出了这些数字的相对大小。图 7.11 中的表的右下角的数字已难以理解：如果宇宙中的每个电子（它们共有 10^{50} 个）都是一台超级计算机，每秒能做亿次（10^8）操作。则自创世纪起（可能为 300 亿年，或大约 10^{18} 秒），至今只完成 10^{76} 次操作，而对要求的 2^{1000} 次操作的计算仍需要

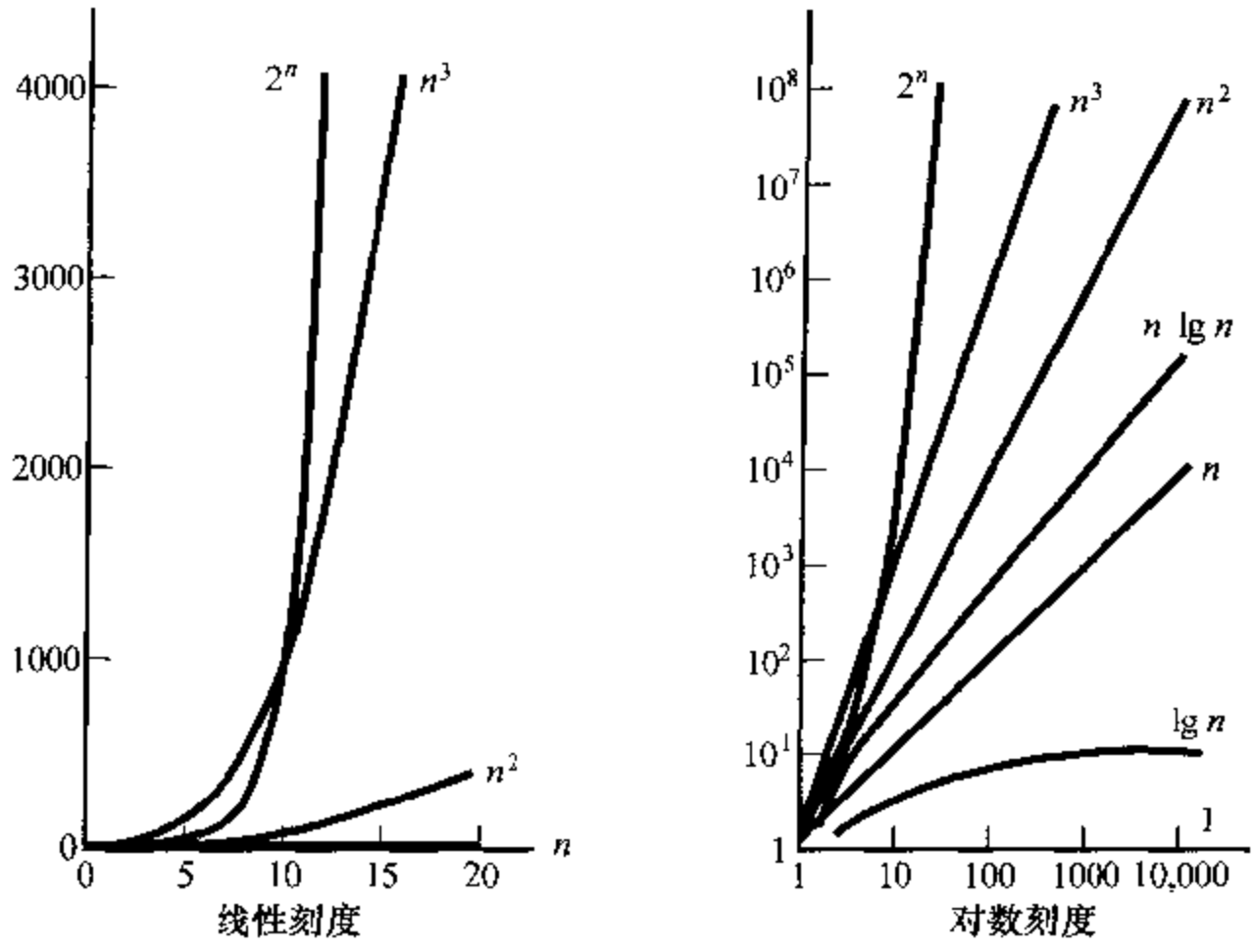


图 7.10 通用函数的增长速率

10^{225}次！因此，要求 2^n 次操作的计算仅对很小的 n 值是可行的。

n	1	$\lg n$	n	$n\lg n$	n^2	n^3	2^n
1	1	0.00	1	0	1	1	2
10	1	3.32	10	33	100	1000	1024
100	1	6.64	100	664	10,000	1,000,000	1.268×10^{30}
1000	1	9.97	1000	9970	1,000,000	10^9	1.072×10^{301}

图 7.11 函数的相对大小

特别注意 $\lg n$ 如何比 n 增长地更慢，这就是为何对于较大的表，二分查找比顺序查找更优的根本原因。注意对较大的 n，函数 1 和 $\lg n$ 如何越来越位于所有其他的函数之下。

7.6.3 大-O和相关的符号

为了在计算中有效地利用数量级，需要一个更简洁的符号，而不是写成“比……有严格小的数量级”这样的用语。

我们要像使用标准的比较符号$<$，$\leqslant$，$=$，$\geqslant$和$>$去比较数字同样的方式来比较数量级。遗憾的是，没有常用的用于数量级的比较符号系统。替代地，可使用某些希腊和拉丁字母的系统，这种系统有其自身的优点，虽然开始时需要一点实践来学习使用它。

下面是 4 种常用的术语：

符号： $f(n)$是	发音： $f(n)$是	意义： 与 g 相比，f 的阶是	$\lim_{n\to\infty}(f(n)/g(n))$的值
$o(g(n))$	$g(n)$的小 oh	<严格小于	0
$O(g(n))$	$g(n)$的大 Oh	≤ 小于或等于	有限的
$\Theta(g(n))$	$g(n)$的大 Theta	=等于	非零有限的
$\Omega(g(n))$	$g(n)$的大 Omega	≥ 大于或等于	非零的

为了详细说明如何使用这些符号，让我们用新的符号来重新进行一些曾经做过的算法分析：

- 在长度为 n 的表上，顺序查找的运行时间为 $\Theta(n)$。
- 在长度为 n 的有序表上，二分查找的运行时间为 $\Theta(\log n)$。
- 从长度为 n 的顺序表上检索的运行时间为 $O(1)$。
- 从长度为 n 的链表上检索的运行时间为 $O(n)$。
- 任何使用键比较查找长度为 n 的表的算法必须进行 $\Omega(\log n)$次键比较(定理 7.6)。
- 汉诺塔(见 5.1.4 节)的任何算法要求时间 $\Omega(2^n)$去移动 n 个盘子。

我们关于数量级所做出的一般观察也能重述如下：

* * * 多项式

- 若 $f(n)$是 n 的 r 次多项式，则 $f(n)$为 $\Theta(n^r)$。

* * * n 次幂

- 若 $r<s$，则 n^r是 $o(n^s)$。

* * * 底数的变化

- 若 $a>1$ 且 $b>1$，则 $\log_a(n)$是 $\Theta(\log_b(n))$。

* * * 对数

- 对任何 $r>0$，$\log n$ 为 $o(n^r)$。

* * * 指数

- 对任何实数 $a>1$ 和任何正整数 r，n^r为 $o(a^n)$。

* * * 两个指数

- 若 $0\leqslant a<b$，则 a^n为 $o(b^n)$。

7.6.4 保留主项

注意，在所有 3 个符号中 O，Θ 和 Ω 中，当极限是有限的非零值时，这些符号对一个非零值和其他值不分彼此，这对算法分析通常比较合适，因为用常数乘以这个结果可能只反映了程序设计风格的变化或计算机速度的变化。

* * * 查找比较

然而，有时我们希望对算法所做的工作量有更精确的度量，这时就可以像下面这样将大-O 使用在表达式中。我们定义 $f(n)=g(n)+O(h(n))$来表示 $f(n)-g(n)$是 $O(h(n))$。对

某个常数 c，不是将 $O(h(n))$ 看作增长不如 $ch(n)$ 快的一类函数，而是将 $O(h(n))$ 看作单个的但却是任意的函数。然后使用这个函数去表达计算中不感兴趣的所有项，一般是除了增长最快的那一项以外的所有项。

我们的一些算法分析的结果现在可以概括如下：

- 对长度为 n 的表的成功查找，顺序查找的运行时间为 $\frac{1}{2}n+O(1)$。
- 对长度为 n 的有序表的成功查找，二分查找的运行时间为 $2\ \lg n+O(1)$。
- 对长度为 n 的顺序表检索的运行时间为 $O(1)$。
- 对长度为 n 的简单链表检索的平均运行时间为 $\frac{1}{2}n+O(1)$。

* * * 危险性

在表达式中使用大-O 符号时，总有必要牢记 $O(h(n))$ 不代表一个准确定义的函数，而是一大类函数中的任意一个函数。因此不能用 $O(h(n))$ 做普通代数。例如我们有两个表达式：

$$n^2+4n-5=n^2+O(n) \text{ 和 } n^2-9n+7=n^2+O(n)$$

但 $O(n)$ 在两个表达式中代表不同的函数，因此不能将两个式子右边等同起来或者得出两个式子左边相等的结论。

练习 7.6

E1. 对下列每对函数，找出 $n>1$ 时第一个函数比第二个函数大时的最小整数值。（对其中一些函数，使用计算器尝试不同的 n 值。）

(1) n^2 和 $15n+5$　　(3) $0.1n$ 和 $10\ \lg n$

(2) 2^n 和 $8n^4$　　(4) $0.1\ n^2$ 和 $100n\ \lg n$

E2. 按阶的升序排列下列函数，即在你的列表中，$f(n)$ 位于 $g(n)$ 之前当且仅当 $f(n)$ 是 $O(g(n))$。

100 000	$(\lg n)^3$	2^n
$n\lg n$	$n^3-100\ n^2$	$n+\lg n$
$\lg\ \lg n$	$n^{0.1}$	n^2

E3. 将下列函数分类，使得两个函数 $f(n)$ 和 $g(n)$ 在同一类中当且仅当 $f(n)$ 是 $\Theta(g(n))$。将这些类从最低的数量级到最高的数量级进行排列。（一个函数可能自己属于一个类，或可能几个函数属于同一类。）

5000	$(\lg n)^5$	3^n
$\lg n$	$n+\lg n$	n^3
$n^2\lg n$	$n^2-100\ n$	$4n+\sqrt{n}$
$\lg\ \lg n^2$	$n^{0.3}$	n^2
$\lg n^2$	$\sqrt{n^2+4}$	2^n

E4. 证明下面每一句话都是正确的。

(1) $3n^2-10\ n\lg n-300$ 是 $\Theta(n^2)$。

(2)$4n\lg n+100n-\sqrt{n+5}$是 $\Omega(n)$。

(3)$4n\lg n+100n-\sqrt{n+5}$是 $o(\sqrt{n^3})$。

(4)$(n-5)(n+\lg n+\sqrt{n}$是 $O(n^2)$。

(5)$\sqrt{n^2+5n+12}$是 $\Theta(n)$。

E5. 判断下面每一句话是否正确。

(1)$(3\lg n)^3-10\sqrt{n}+2n$ 是 $O(n)$。

(2)$(3\lg n)^3-10\sqrt{n}+2n$ 是 $\Omega(\sqrt{n})$。

(3)$(3\lg n)^3-10\sqrt{n}+2n$ 是 $o(n\log n)$。

(4)$\sqrt{n^2-10n+100}$是 $\Omega(n)$。

(5)$3n-10\sqrt{n}+\sqrt{n\lg n}$是 $O(n)$。

(6)2^n-n^3是 $\Omega(n^4)$。

(7)$\sqrt{3n-12n\lg n-2n^3+n^4}$是 $\Theta(n^2)$。

(8)$(n+10)^3$是 $O((n-10)^3)$。

E6. 假设你有一组程序，对长度为 n 的输入，它们的运行时间分别为 $1000\lg n$、$100n$、$10n^2$ 和 2^n。找出下列时间内每个程序所能处理的最大输入长度 n：(1)1 秒；(2)1 分钟；(3)1 天；(4)1 年。

E7. 证明函数 $f(n)$为 $\Theta(g(n))$当且仅当 $f(n)$为 $O(g(n))$且 $f(n)$为 $\Omega(g(n))$。

E8. 假设 $f(n)$为 $\Theta(g(n))$且 $h(n)$为 $o(g(n))$，证明 $f(n)+h(n)$为 $\Theta(g(n))$。

E9. 找出函数 $f(n)$和 $h(n)$，它们都是 $\Theta(n)$，但 $f(n)+h(n)$不是 $\Theta(n)$。

E10. 假设 $f(n)$是 $O(g(n))$且 $h(n)$是 $O(g(n))$，证明 $f(n)+h(n)$为 $O(g(n))$。

E11. 证明关系 O 是传递的，即从假设 $f(n)$为 $O(g(n))$且 $g(n)$为 $O(h(n))$，证明 $f(n)$为 $O(h(n))$。任何其他的关系 o、Θ 和 Ω 是传递的吗？若是，则是哪一个？

E12. 证明关系 Θ 是对称的，即从假设 $f(n)$为 $\Theta(g(n))$，证明 $g(n)$为 $\Theta(f(n))$。任何其他的关系 o、O 和 Ω 是对称的吗？若是，则是哪一个？

E13. 证明关系 Ω 是自反的，即证明任何函数 $f(n)$为 $\Omega(f(n))$。任何其他的关系 o、O 和 Θ 是自反的吗？若是，则是哪一个？

E14. 如果一个关系是自反的、对称的和传递的，则称此关系为等价关系。在前述 3 道练习的基础上，指出 Θ、o、O 和 Ω 中哪一个是等价关系？

E15. 假设你通过不看代码而运行程序的方法来评价计算机程序，用几个输入长度 n 来运行每个程序并得到所示的操作次数或时间。在每种情况下，基于给出的数字，找出一个常数 c 和一个函数 $g(n)$(它应是图 7.10 中所示的 7 个函数之一)，使得 $cg(n)$几乎近似于给出的数字。

(1)n：	10	50	200	1000
次数：	201	998	4005	19987
(2)n：	10	100	1000	10000
次数：	3	6	12	24
(3)n：	10	20	40	80

次数：	10	40	158	602
(4)n：	10	11	12	13
次数：	3	6	12	24

E16. 如果两个函数 $f(n)$和 $g(n)$有$\lim_{n\to\infty}\frac{f(n)}{g(n)}=1$，则称这两个函数渐近相等(asymptotically equal)，这种情况我们写作 $f(n)\asymp g(n)$。证明下面每一个成立：

(1)$3x^2-12x+7\asymp 3x^2$

(2)$\sqrt{x^2+1}\asymp x$

(3)若 $f(n)\asymp g(n)$，则 $f(n)$为 $\Theta(g(n))$。

(4)若 $f(n)\asymp g(n)$且 $g(n)\asymp h(n)$，则 $f(n)\asymp h(n)$。

程序设计项目 7.6

P1. 编写程序在你的计算机上测试当 $n=5,10,15$ 和 20 时，需要多长的时间计算 $n\lg n$，n^2，n^5，2^n 及 $n!$ 的和。

启示和易犯的错误

1. 在设计算法时，小心极端情况，如空表、仅有一个元素的表或满表(在顺序情况下)。
2. 确信所有变量被正确地初始化。
3. 加倍努力地检查循环的终止条件，并确信循环总会向终止进展。
4. 万一有困难，明确地表达出在循环的每次重复前后都正确的语句，并验证它们成立。
5. 避免为了复杂而复杂，如果简单的方法已足够于你的应用，则使用简单的方法。
6. 避免做重复的工作，如果现成的函数已足够于你的应用，则使用现成的函数。
7. 顺序查找虽然较慢但却很健壮，在短表或未知表中的键是否合适排序的情况下使用顺序查找。
8. 如果必须对二分查找重新进行程序设计，则务必特别小心，验证你的算法正确，并在所有极端情况下测试它。
9. 画树是一种优秀的方法，可以同时用于跟踪算法的活动和分析它的行为。
10. 对大型应用依赖于算法的大一O 进行分析，但对小型应用则不要。

复习题

7.4

1. 列举表的顺序查找比二分查找更可取的 3 个条件。

7.5

2. 在查找 n 个元素的表时，下面的函数平均各需进行多少次键比较：(1)sequential_search；(2)binary_search_1；(3)binary_search_2?

第8章　排　　序

本章研究几种用于顺序表和链表排序的重要方法，同时将设计更多的有助于算法分析的工具，并且应用这些工具去确定不同情况下哪一种排序方法的性能更好。

8.1　引言和符号

我们生活在一个被信息保存困扰的世界中，为了查找信息，我们必须以某种切合实际的顺序保存它。图书馆员确保大家不会误放了书籍，纳税管理机构明白我们所挣的每一块钱，信用社知道我们行动的几乎每个细节。我曾经看过一个动画，其中有一名热心的档案管理员，急于给老板留下印象，狂热地说："在我们丢弃这些文件前，我确信它们按照字母顺序排列。"如果我们想成为这种知识爆炸的主人而不是受害者的话，最好还是学会如何明了全部的知识。

*** 实际重要性

几年前，据估计许多商业计算机有超过一半的时间是花在排序上。或许这不再是事实，因为现在已经设计出更高级的组织数据的方法，这些方法不要求数据必须按任何特定的顺序保存。尽管如此，最终信息还是会到人那里，因而必须以某种方式对它进行排序。

*** 外部排序和内部排序

由于排序是如此重要，所以迄今已设计出许多种排序算法。实际上，在排序方法中产生了许多优秀的思想，以至于可以围绕这个主题设置一门课程。在不同的环境中需要不同的方法，而最重要的是认识到内部和外部的区别，即：是否有太多的数据需要排序，使得必须将它们保存在磁盘、磁带等介质上的外部文件中，或者是否能够将它们保存在高速内存中。本章只考虑内部排序。

*** 参考文献

我们的意图并不是要提供有关内部排序方法的近乎全面的讨论，相关讨论可以参见 D. E. KNUTH 里程碑之作的第 3 卷（在第 2 章结尾给出的参考文献），KNUTH 讨论了大约 25 种排序方法，并断言它们"只是迄今为止所设计的算法中的一小部分。"我们只详细地研究其中的几个方法，之所以选择它们是因为：

- 它们是优秀的——每一种方法都是某种情况下的最佳选择。
- 它们阐述了全部方法中出现的许多变种。
- 它们相对易于编写和理解，不需要太多的细节而使表示复杂化。

在练习中将出现这些方法的许多变种。

*** 符号

贯穿本章都将使用第 6 章和第 7 章中建立的符号，因此我们将按照与记录相关的键所决定的顺序对记录的表进行排序。表的定义和所分配的各种类型和操作的名称将与前面章节相同。

有时有一种情况在练习时必须特别小心：表中两个或更多个元素可能具有相同的键。在这种有重复键的情况下，对具有重复键的元素排序可能会产生不同的顺序。如果具有重复键的元素的顺序对一个应用有所区别的话，那么在构造排序算法时必须特别小心。

＊＊＊基本操作

在研究查找算法时，我们很快就弄清楚工作总量与键比较的次数密切相关。同样的结果对排序算法亦成立，但排序算法必须做到：或者修改指针，或者在表中来回移动元素，因此这样花费的时间也很重要，特别是在顺序表中保存的是较大元素的情况下，因此我们的分析将集中考虑这两种基本的操作。

＊＊＊分析

像以前一样，排序算法最坏情况下的性能和平均性能都是我们的兴趣所在。为找出平均性能，我们要考虑算法在表的所有可能的排列上(这个表有 n 个元素，因此共有 n! 种这样的排列)运行时所发生的情况，然后取这些结果的平均值。

8.1.1 可排序的表

贯穿本章，我们将特别关心排序算法的性能。为了优化对表进行排序的程序的性能，需要利用表的实现所具有的任何特殊性质。例如我们会看到某些排序算法在顺序表中运行起来特别有效，但对链表的有效排序却需要不同的实现和不同的算法。因此，为了编写有效的排序算法，我们需要访问待排序表的私有数据成员，为此要将排序函数作为基本的 List 数据结构的方法加入。增大后的表结构形成了一种新的 ADT，我们称之为 Sortable_list，它的类定义采取如下的形式：

```
template <class Record>
class Sortable_list: public List <Record>{
public:                                  //  Add prototypes for sorting methods here.
private:                                 //  Add prototypes for auxiliary functions here.
};
```

这种定义表明 Sortable_list 是一个带有附加排序方法的 List。同往常一样，类中用于创建此方法的辅助函数是客户程序代码不可用的函数，基表类可以是第 6 章中 List 的任何一种实现。

＊＊＊Record 和 Key

我们使用称为 Record 的模板参数类表示 Sortable_list 中的元素，与第 7 章中一样，我们假设 Record 类具有下列性质：

＊＊＊需求

每个 Record 有一个关联的 Key 类型的键。Record 能被隐含地转换成其对应的 Key，此外，键(因此也是记录)能够用运算符‘<’，‘>’，‘>=’，‘<=’，‘==’和‘!=’进行比较。

通过将一个客户程序作为 Sortable_list 的模板参数，可以提供第 7 章讨论的任何

Record 的实现。例如用于测试我们的 Sortable_list 的程序可以简单地定义为：Sortable_list <int>test_list;。这里，客户程序使用 int 类型同时表达记录和它们的键。

8.2 插入排序

8.2.1 有序插入

当最初在 7.3 节中介绍二分查找时，曾提到有序表(ordered list)只是一种新的抽象数据类型，我们将它定义为一种表，表的每个元素有一个键，并且这些键是有序的，即：如果在表中元素 i 位于元素 j 之前，则元素 i 的键小于等于元素 j 的键。我们假设这些键能用运算符'<'和'>'进行比较(例如，键可以是数字，或者是重载了比较运算符的类的实例)。

对有序表，经常会用到两个新的操作，其他的表没有与之对应的操作，因为它们是利用键而不是利用位置来查找元素。

*** 按键检索

- 一种操作是从有序表中检索具有指定键的元素。

*** 按键插入

- 第二种操作向有序表中插入一个新元素，利用新元素中的键确定应插入到表中的位置。

注意，如果表中已包含有与新元素具有同样键的元素，则插入不是惟一指定的，因为新元素能放在多于一个位置上。

从有序表中按键检索确实与查找是一样的，我们已在第 7 章研究过这个问题。有序插入将充当我们的第一个排序方法的基础。

*** 顺序表的有序插入

首先，我们考虑顺序表。在此情况中，必须移动表中的元素为插入腾出空间。为了找到插入的位置，必须进行查找。在顺序表中执行有序插入的一种方法是先进行二分查找以找到正确的位置，然后按需移动元素并插入新元素，这种方法将留作练习。由于不论查找如何进行，都需要大量的时间移动元素，事实证明在许多情况下，使用顺序查找同二分查找同样快。通过从表尾开始进行顺序查找，可以将这个查找过程与元素的移动结合在一个单循环中，因此降低了函数所需的开销。

*** 例子

有序插入的一个例子见图 8.1，我们从图的(a)部分所示的有序表开始，希望插入新元素 hen。与 7.3 节的独立实现的 insert 版本相对比，我们在表尾而非表头开始进行键比较，因此我们首先将新键 hen 同浅灰色框中显示的最后一个键 ram 比较，因为 hen 在 ram 前，因此将 ram 往下移，留出(b)部分所示的空位置。接下来将 hen 与(b)部分浅灰色框中显示的键 pig 比较，hen 再次位于前，因此将 pig 往下移，并将 hen 与(c)部分所示的浅灰色框中显示的键 dog 比较，由于 hen 在 dog 之后，我们已找到合适的位置，因此如(d)部分所示完成插入。

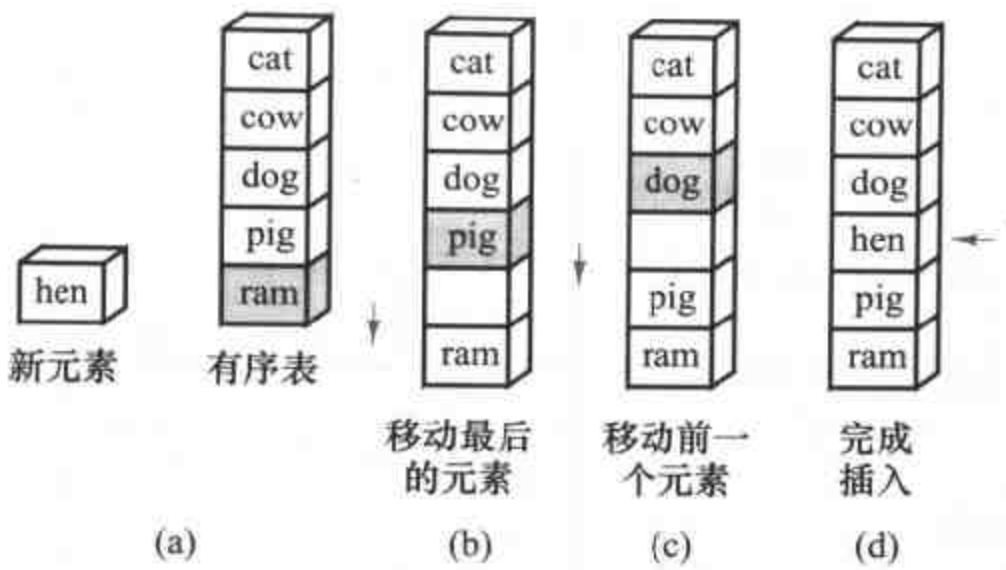


图 8.1　有序插入

8.2.2　通过插入进行排序

第一种表排序方法基于向有序表中插入的思想。为了对一个无序表进行排序，考虑每次移动其中一个元素，并随后将每个元素插入到一个初始为空的新表中，并总是保持新表中的元素按照它们的键有合适的顺序。

＊＊＊例子

图 8.2 中给出了这种方法的图解说明，它展示了对 6 个单词的表进行排序所需要的步骤。在每个阶段，尚未插入到有序表中的单词显示在浅灰色框中，表的已排序部分显示在白色框中。在初始的图中，第一个单词 hen 显示为有序的，因为长度为 1 的表是自动有序的，所有的剩余单词在此阶段显示成未排序的。在此过程的每一步，第一个未排序的单词（显示在最上面的灰色框中）被插入到表的有序部分的合适位置处。为了给插入腾出空间，某些已排序的单词必须在表中向下移动，单词的每次移动在图 8.2 中显示成一个浅灰色的箭头。通过在表的有序部分的尾部开始，我们能在比较的同时移动元素以找到新元素的合适位置。

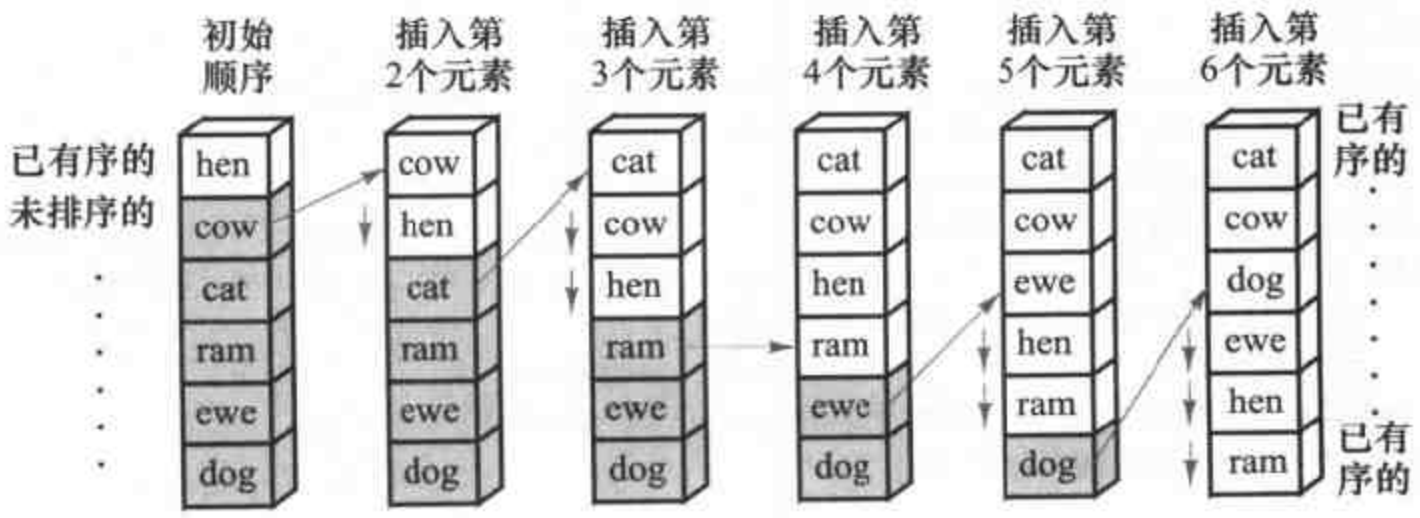


图 8.2　插入排序的例子

向表的有序部分中插入用 current 指示的元素所需的主要步骤如图 8.3 所示。在接下来的方法中，我们假设 Sorted_list 类基于 6.2.2 节顺序 List 的实现。有序表和未排序表都

同时合用相同的成员数组 List，记得在 6.2.2 节称这些成员为元素。变量 first_unsorted 标记这个数组的有序部分和未排序部分之间的分割。现在我们编写这个算法。

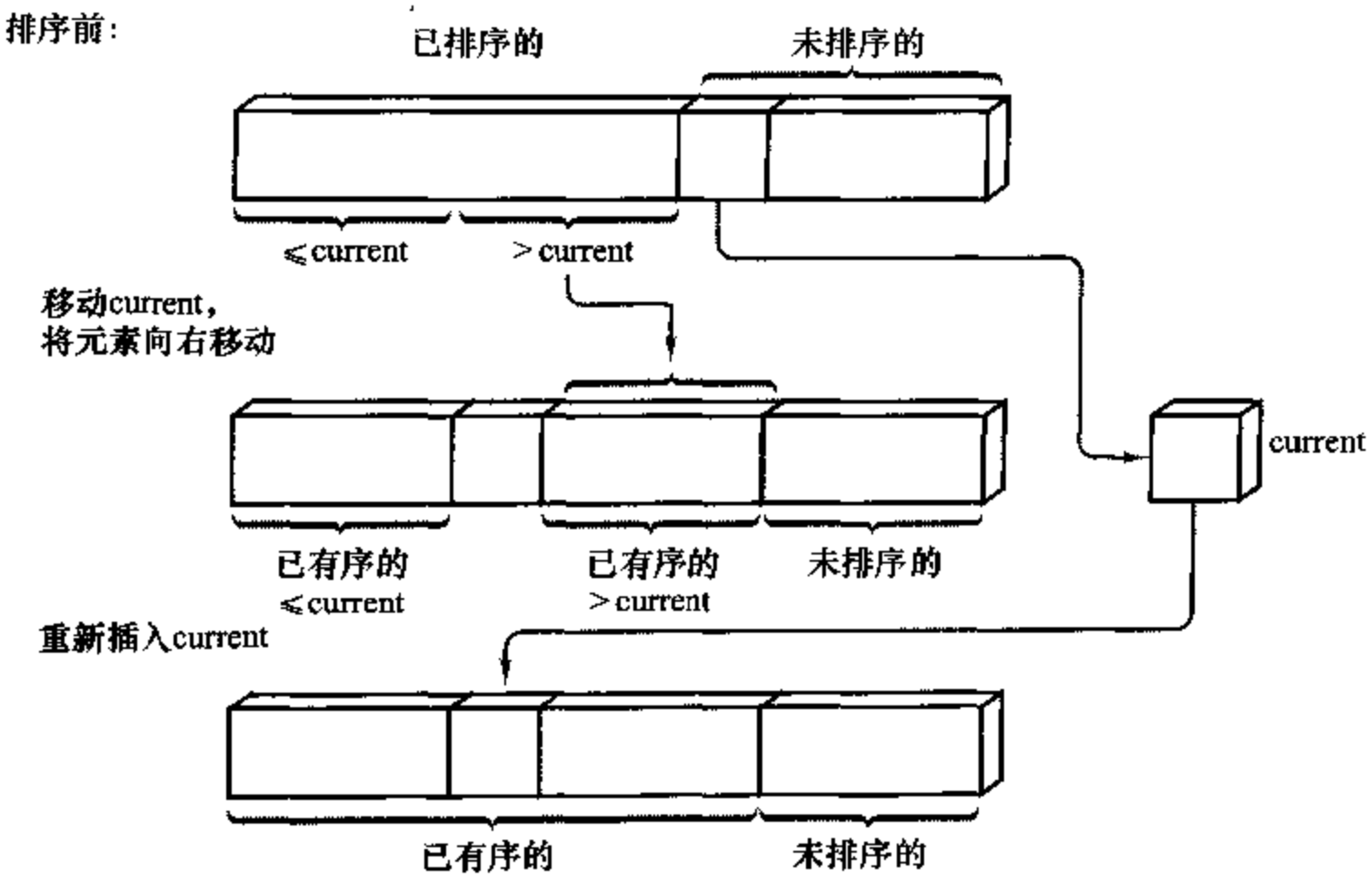


图 8.3 顺序插入排序的主要步骤

```
template <class Record>
void Sortable_list <Record>:: insertion_sort()
/*Post: The entries of the Sortable_list have been rearranged so that the keys in all the
        entries are sorted into nondecreasing order.
  Uses: Methods for the classRecord; the contiguous List implementation of Chapter 6* /

{
  int first_unsorted;          //  position of first unsorted entry
  int position;                //  searches sorted part of list
  Record current;              //  holds the entry temporarily removed from list
  for (first_unsorted=1; first_unsorted<count; first_unsorted++)
    if (entry [first_unsorted]<entry[first_unsorted-1]){
      position=first_unsorted;
      current=entry[first_unsorted];   //  Pull unsorted entry out of the list.
      do {          //  Shift all entries until the proper position is found.
        entry[position]=entry[position-1];
        position--;                 //position is empty.
      }while (position>0 && entry[position-1]>current);
      entry[position]=current;
  }
}
```

这个程序的操作一目了然。由于只有一个元素的表是自动有序的，因此 first_unsorted 上的循环从第二个元素开始，如果它处于正确的位置，则不用做任何事情，否则需要将新元素从表中取出并放入变量 current 中，然后 do … while 循环将各个元素往后移一个位置，直

到找到正确的位置，最后将 current 插入其中，并继续处理下一个未排序的元素。current 属于表中第一个位置的情况必须特别检测，在这种情况中，表中没有更小键的元素去终止查找过程，因此我们在 do … while 循环条件的第一个语句中处理这种情况。

8.2.3 链式版本

* * * 算法

对插入排序的链式版本来说，由于不需要移动数据，就不必在已排序的子表的尾部开始查找。相反，我们将遍历原始的表，每次取一个元素并将它插入到有序表中的合适位置。指针变量 last_sorted 将指向表的已排序部分的尾部，last_sorted->next 将指向尚未被插入已排序子表的第一个元素。我们让 first_unsorted 也指向这个元素，并用指针 current 查找表的已排序部分，找到插入 * first_unsorted 的位置。如果 * first_unsorted 位于当前的表头之前，则将它插入到那里，否则将 current 在表中一路后移，直到 first_unsorted->entry<= current->entry，再将 * first_unsorted 插入到 * current 之前。为了能在 * current 之前插入，我们用一个附加的指针 trailing 紧跟着 current 并比 current 靠近表头一个位置。

* * * 停止循环

监视哨(sentinel)是加入到表的一端的附加元素，其目的是不必包含单独的检查而能确保循环终止。由于有 last_sorted->next=first_unsorted，结点 * first_unsorted 已经处于充当查找的监视哨的位置上，因此简化了移动 current 的循环。

最后，我们指出具有 0 或 1 个元素的表已经有序，因此我们能单独地检查这些情况，从而避免了过程繁琐。下面的函数给出了细节，并在图 8.4 中给予说明。

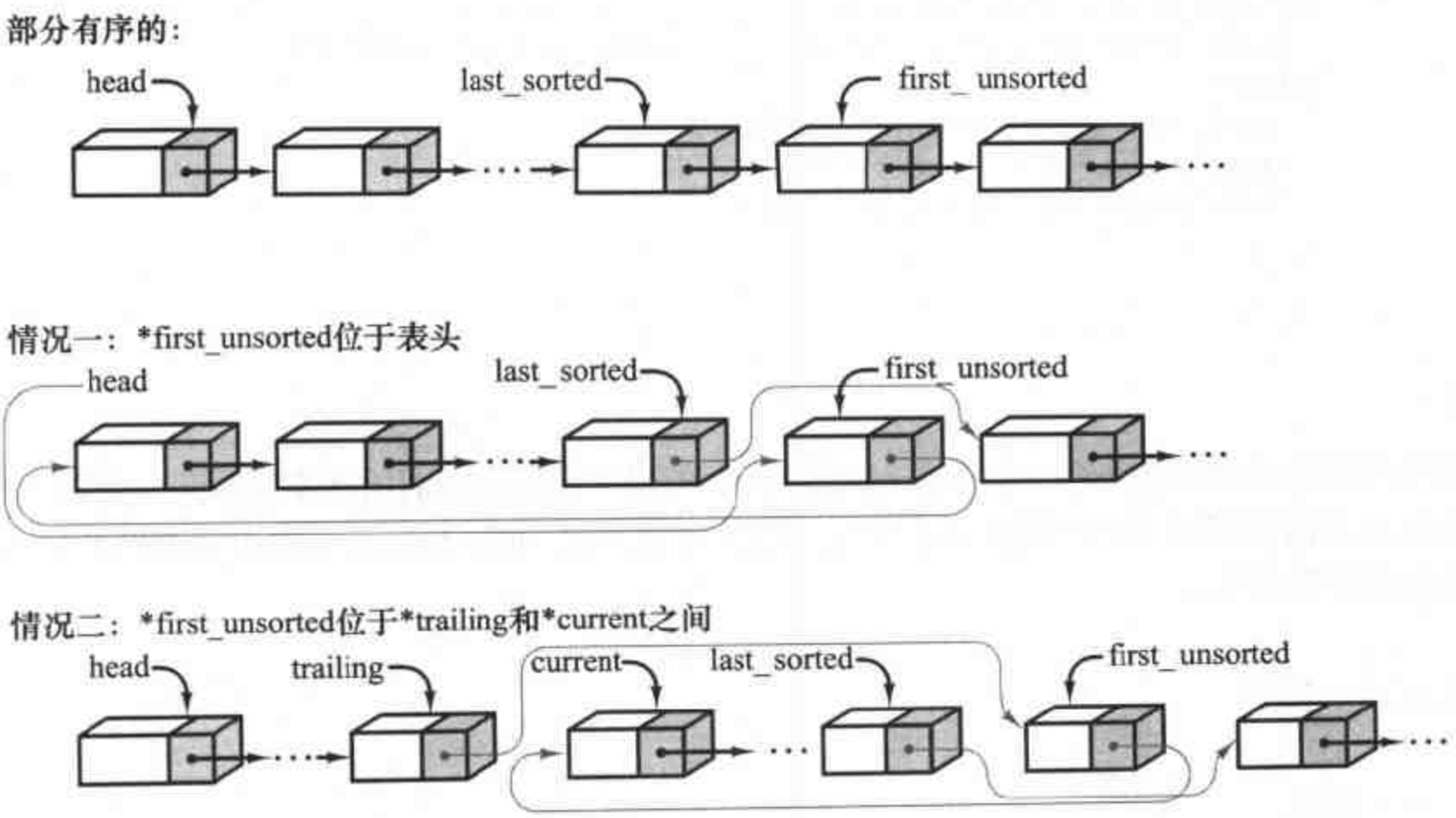


图 8.4 链式插入排序的描绘

```
template <class Record>
void Sortable_list<Record>::insertion_sort()
/* Post: The entries of the Sortable_list have been rearranged so that the keys in all
         the entries are sorted into nondecreasing order.
   Uses: Methods for the class Record, The linked list implementation of Chapter 6. */
{
  Node<Record>* first_unsorted,     //  the first unsorted node to be inserted
              * last_sorted,        //  tail of the sorted sublist
              * current,            //  used to traverse the sorted sublist
              * trailing;           //  one position behind current
  if (head!=NULL){                  //  Otherwise, the empty list is already sorted.
    last_sorted=head;               //  The first node alone makes a sorted sublist.
    while (last_sorted->next!=NULL){
      first_unsorted=last_sorted->next;
      if(first_unsorted->entry<head->entry){
        //  Insert *first_unsorted at the head of the sorted list:
        last_sorted->next=first_unsorted->next;
        first_unsorted->next=head;
        head=first_unsorted;
      }
      else {
        //  Search the sorted sublist to insert *first_unsorted:
        trailing=head;
        current=trailing->next;
        while (first_unsorted->entry>current->entry){
          trailing=current;
          current=trailing->next;
        }
        // * first_unsorted now belongs between *trailing and *current.
        if (first_unsorted==current)
          last_sorted=first_unsorted;//  already in right position
        else {
          last_sorted->next=first_unsorted->next;
          first_unsorted->next=current;
          trailing->next=first_unsorted;
        }
      }
    }
  }
}
```

即使链式版本的结构与顺序版本有很大的不同，但仍可以看出它们的基本方法是一致的，仅有的真正的区别在于顺序版本以逆序查找已排序的子表，而链式版本以表中位置的升序查找已排序的子表。

8.2.4 分析

* * * 假设

由于基本思想是一样的，我们只分析顺序版本的程序的性能，同时仅将注意力限于表最初为随机顺序（意味着所有可能的键序具有同等可能性）的情况。当处理元素 i 时，究竟必须往回走多远才能插入它？有 i 种方式去移它：完全不移、移动一个位置和一直向表的前部

移动 $i-1$ 个位置。由于已假设了随机性，这 i 种方式具有同等可能，不必移动的概率因此为 $1/i$，这种情况下只进行一次键比较，不用移动元素。

＊＊＊插入一个元素

其余的情况都必须移动元素 i，发生的概率为 $(i-1)/i$。我们从计算 do … while 循环的平均重复次数开始。由于所有的 $i-1$ 个可能的位置具有同等可能，因此循环的平均次数为：

$$\frac{1+2+\cdots+(i-1)}{i-1}=\frac{(i-1)i}{2(i-1)}=\frac{i}{2}$$

（此计算利用了附录 A 的定理 A.1。）其中的每一次重复要进行一次键比较和一次赋值，另外在循环外多进行一次键比较和两次元素赋值，因此在第二种情况中，每个 i 平均需要 $\frac{1}{2}i+1$ 次比较和 $\frac{1}{2}i+2$ 次赋值。

结合这两种情况各自的规律，比较次数共有：

$$\frac{1}{i}\times 1+\frac{i-1}{i}\times\left(\frac{i}{2}+1\right)=\frac{i+1}{2}$$

赋值次数共有：

$$\frac{1}{i}\times 0+\frac{i-1}{i}\times\left(\frac{i}{2}+2\right)=\frac{i+3}{2}-\frac{2}{i}$$

＊＊＊插入所有的元素

我们希望从 $i=2$ 到 $i=n$ 累加这些数，但为了避免运算的复杂性，我们首先利用大一 O 符号（见 7.6.3 节），通过消除那些受常数约束的项（即为 $O(1)$ 的项），对每个表达式取近似。因此对元素的比较次数和赋值次数都得到 $\frac{1}{2}i+O(1)$。在取近似时，我们实际上注意主循环中的操作并且不关心循环外部的操作或算法中的变量，因为它们有限改变工作量。

为了从 $i=2$ 到 $i=n$ 累加 $\frac{1}{2}i+O(1)$，我们应用附录 A 的定理 A.1（从 1 到 n 的整数和）。我们同时指出，将 n 个项相加，其中每个项为 $O(1)$，则产生的结果为 $O(n)$，因此得到键比较次数和元素赋值次数均为：

$$\sum_{i=2}^{n}\left(\frac{1}{2}i+O(1)\right)=\frac{1}{2}\sum_{i=2}^{n}i+O(n)=\frac{1}{4}n^2+O(n)$$

迄今为止，还没有什么东西可以用来比较这个数，但能够注意到的是，当 n 变大时，含有 n^2 的项的作用比其余汇总成 $O(n)$ 的项大得多。因此当表的长度增长时，插入排序所需要的时间的增长如同表长度的平方。

＊＊＊最佳和最坏情况

插入排序的最坏情况的分析将留作练习。我们注意到顺序插入排序的最佳情况出现在表已有序时，此时插入排序只需进行 $n-1$ 次键比较即可。现在可以指出的是：最佳情况下，其他的排序方法都不会比顺序插入排序更好。

定理 8.1　验证以恰当次序排列的 n 个元素的表要求至少 $n-1$ 次键比较。

证明 考虑检查 n 个元素的表是否有序(若不是,则对它进行排序)的任意程序。此程序首先要进行一些键比较,这些比较会涉及表中的某两个元素。在随后的某个时间,这两个元素中至少有一个会与第三个元素比较,否则将无法确定这两个元素在表中相对于第三个元素的位置。因此,第二次比较仅仅包含一个先前不在比较中的新元素。如此继续下去,我们发现必然会出现这前面二个元素中的某个与一个新元素的另一次比较。注意我们不必按算法执行比较的次序选择这些比较。因此,除了第一次比较以外,我们所选择的每次比较都仅涉及一个以前未比较过的新元素。除非将一个元素至少与另一个其他元素比较,否则无法确定这个元素是否处于合适的位置,故所有的 n 个元素都要参与某次比较。因此为了包含所有 n 个元素,需要至少 $n-1$ 次比较。(**证明结束**)

用这个定理,可以发现插入排序的一个优点:它证实插入排序对正确排过序的表排序最快。此外,只要表是近乎有序的,或者表中只有很少的元素离其正确的位置较远,插入排序仍是一种极好的方法。

练习 8.2

E1. 手工跟踪在下列每个表上使用插入排序的步骤,在每种情况中,计算需要的比较次数和移动元素的次数。

(1)下列 3 个单词按字母顺序排序:

triangle square pentagon

(2)对(1)中的 3 个单词按对应的多边形的边数,以升序排序。

(3)对(1)中的 3 个单词按对应的多边形的边数,以降序排序。

(4)将下列 7 个数以升序排序:

26 33 35 29 19 12 22

(5)同样的 7 个数以不同的初始顺序,再次以升序排序:

12 19 33 26 29 35 22

(6)将下列 14 个名字按字母顺序排序:

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

E2. 对顺序表的插入排序来说,键的初始次序如何将产生最坏情况?对链表的插入排序又如何?

E3. 顺序插入排序在其最坏情况下要进行多少次键比较和元素赋值?

E4. 修改链式版本的插入排序,使得已经有序或近乎有序的表能被迅速处理。

程序设计项目 8.2

* * * 排序的测试程序

P1. 编写一个能用于测试和评价插入排序及其他排序方法的程序,使用下面的要点。

(1)创建一些整数文件以用于测试排序方法。创建的这些文件要有不同的长度,如长度分别为 20,200 和 2000,以顺序、逆序、随机顺序或部分有序的方式创建这些文件。

通过将所有这些测试数据保存在文件中(而不是每次在测试程序运行时用随机数生成它),可以使用同样的数据去测试不同的排序方法,因此会更易于比较这些方法的性能。

(2)为测试各种排序方法编写一个菜单驱动程序,其中的一个选项是将一个整数文件读入表中,其他的选项包括:在这个表上运行多种排序方法中的一种;打印出未排序或已排序的表;以及退出程序。在对这些表进行排序和(可能的)输出后,需要废除这个表,以使得后面的排序方法的测试仍用同样的输入数据进行。完成这个功能的方法可以将未排序的表复制到一个附加的表中并对那个表排序,或者也可以安排程序在每次开始排序前重新从数据文件读入数据。

(3)在程序中包含代码去计算和打印(1)CPU 时间,(2)键比较次数和(3)表排序期间表元素的赋值次数。如 7.2 节,可以通过重载 Key 类的比较运算符使得它们递增一个计数器,从而实现对比较次数计数。用类似的方式,能重载 Record 类的赋值运算符以保持对元素赋值的计数。

(4)使用 6.2.2 节中开发的顺序表软件包,加入顺序版本的插入排序,汇集顺序插入排序的性能统计数字,用于以后同其他方法进行比较。

(5)使用 6.2.3 节中开发的顺序表软件包,加入链式版本的插入排序,汇集其性能统计数字,并与顺序插入排序进行比较。为何此版本的元素赋值次数不怎么重要?

* * * 二分插入排序

P2. 重新编写顺序版本的 insertion_sort 函数,使它使用二分查找去定位在何处插入下一个元素。与原始函数 insertion_sort 比较排序一个表所需要的时间。在链式版本的 insertion_sort 中使用二分查找合理吗?为什么?

* * * 扫描排序

P3. 存在一种更容易的排序方法,它不是使用两个指针在表上移动,而是仅使用一个指针。我们可称之为扫描排序(scan sort),它从表的一端开始并向前移动,比较毗邻的键对,直到发现一对失序的键。然后它就交换这对元素,并换一个方向移动,继续交换键对,直到发现一对正确次序的键。此时,它知道已将一个元素根据需要尽可能地后移,因此表的前面部分是有序的,与插入排序不同,它未记住已往前排了多远,因此仅是简单地转变方向并再次往前查找失序的键对,并进行排序。当它到达表的远端时,排序结束。

(1)编写C++程序实现顺序表上的扫描排序,程序中应该仅使用一个位置变量(不同于表的 count 成员)和一个用于交换目的的 entry 类型的变量,而不再使用其他的局部变量。

(2)比较你的程序与使用 insertion_sort 的程序的计时。

* * * 冒泡排序

P4. 一个著名的算法称为冒泡排序,它从左至右对表进行扫描,只要发现一对毗邻的键是失序的,则交换那些元素。在此过程的第一遍中,表中最大的键会被"冒泡"到尾端,但尾端之前的键仍是失序的。因此对失序键对扫描的趟数放在一个循环中,第一趟循环扫描的次数与 count 完全一致,且循环的每次重复都比上一次提前一个位置停止。(1)为冒泡排序编写C++函数。(2)在各种表中找出冒泡排序的性能,并将它与插入排序进行比较。

8.3 选择排序

插入排序有一个主要的缺点，就是即使大多数元素在表的开始部分已被适当地排过序，后面的元素插入仍会要求移动其中的许多元素。插入排序所做的所有移动只是每次移动一个位置，因此为了将一个元素在表中往上移 20 个位置将需要 20 次独立的移动。如果元素比较小，或者只是一个键，或者元素是以链式存储的，则许多次移动不需要过多的时间。但如果元素非常大，比如人事文件或学生成绩单这样的记录会包含几百个分量，并且这些记录必须以顺序存储去保存，则当必须移动元素时，如果能将元素立即移到它最终的位置上，这样会有效地多。下一种排序方法即实现这个目标。

8.3.1 算法

这种排序方法的一个例子在图 8.5 中给出，它给出了按字母顺序对 6 个单词进行排序所需要的步骤。在第一个阶段，对表进行扫描以找到按字母顺序排在最后的单词，这个单词是 ram，显示在浅灰色的框中。然后将这个单词与最后位置上的单词交换，如图 8.5 的第二部分所示。现在再对省略了最后元素而得到的缩短的表重复这个过程，再次排在最后的单词显示在浅灰色的框中，并将它与仍在考虑中的最后的元素交换。如此继续下去。在每个阶段，除了将排在最后的那个元素显示在浅灰色框中以外，所有其他尚未排序的单词都显示在灰色框中。当未排序的表的长度减到 1 时，这个过程结束。

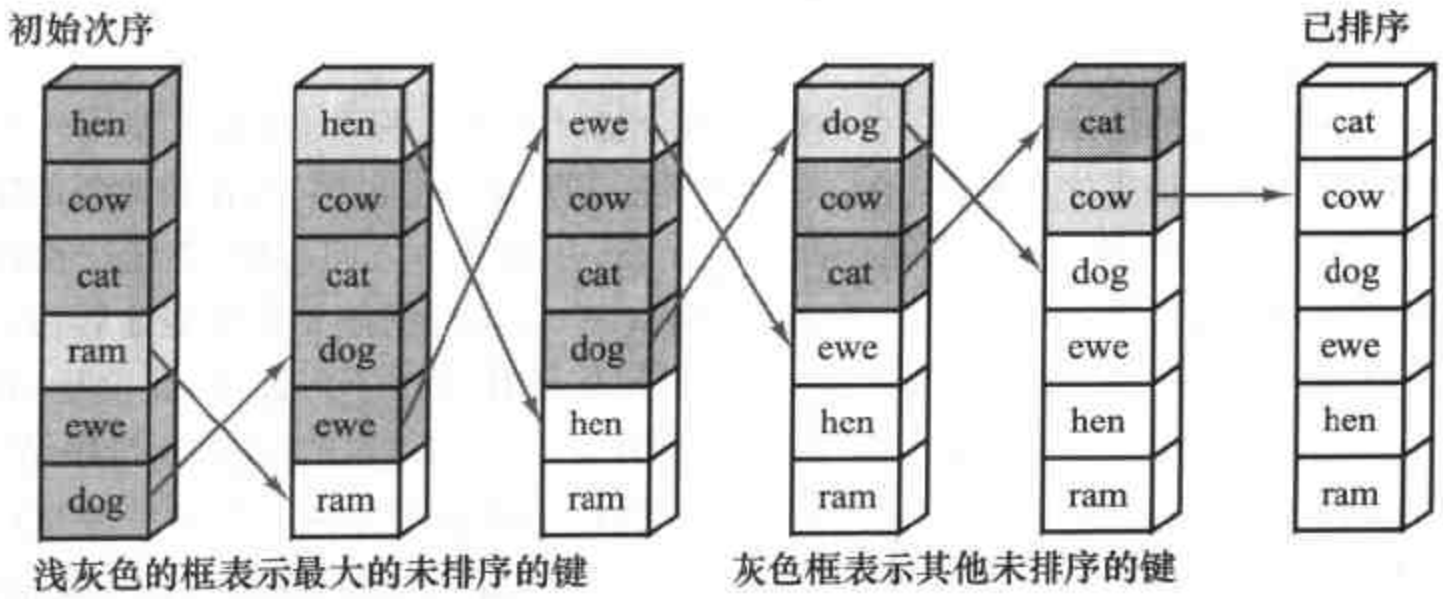


图 8.5 选择排序的例子

这种方法转换成一种称为选择排序(selection sort)的算法，图 8.6 中说明了选择排序的一般步骤。具有较大的键的元素将被排序并放在表的尾端，具有较小键的元素尚未排序。然后我们从前往后查看全部未排序的元素，从中找出具有最大键的那一个，并将它与最后的未排序的元素交换位置。这样，在主循环的每一趟，都有一个元素会被放到其最终的位置上。

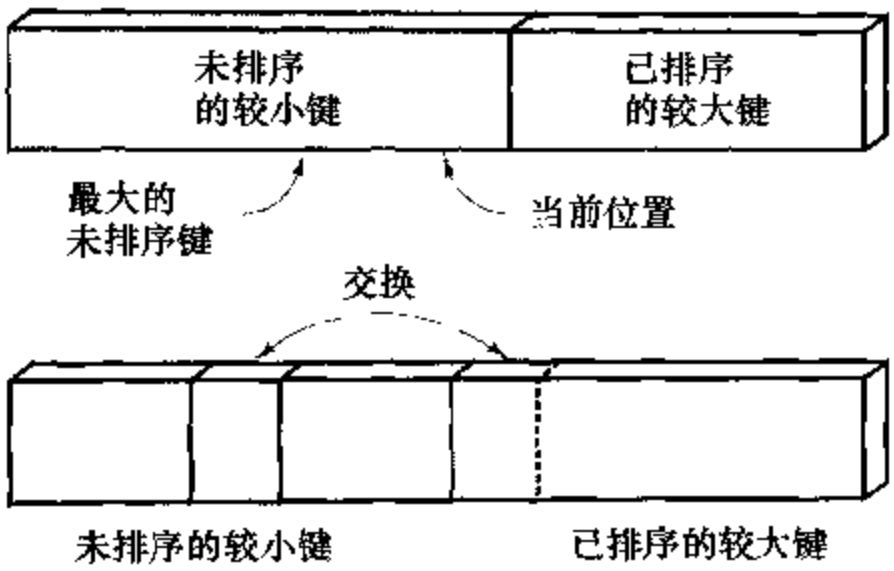


图 8.6　选择排序的一般步骤

8.3.2　顺序实现

选择排序在每一趟都会将至少一个元素放在其最终位置上，从而使数据的移动最少。这个算法主要对大元素的顺序表有用，在这种表中移动元素往往代价太大。如果元素较小，或者表是链式的，在表排序时只需要改变指针，则插入排序一般比选择排序更快，因此我们仅给出选择排序的顺序版本。这个算法使用 Sortable_list 的一个辅助成员函数，称为 max_key，它找出以参数给出的那部分表中的最大的键。辅助函数 swap 仅用于交换给定下标的两个元素。为便于接下来的讨论，我们将这两个函数写成独立的辅助成员函数：

```
template <class Record>
void Sortable_list<Record> :: selection sort()
/* Post: The entries of the Sortable_list have been rearranged so that the keys in all
         the entries are sorted into nondecreasing order.
   Uses: max_key, swap. */
{
   for (int position=count-1; position>0; position--){
      int max=max_key(0, position);
      swap(max, position);
   }
}
```

注意，当表中除一个元素以外，其他元素都位于正确的位置时，那么这一个元素必然也位于正确的位置上。因此 for 循环在 1 上停止。

```
template <class Record>
int Sortable_list<Record> :: max_key(int low, int high)
/* Pre: low and high are valid positions in the Sortable_list and low<=high.
   Post: The position of the entry between low and high with the largest key is returned.
   Uses: The class Record. The contiguous List implementation of Chapter 6. */
{
   int largest, current;
   largest=low;
   for (current=low+1; current<=high; current++)
      if (entry[largest]<entry[current])
```

```
      largest=current;
  return largest;
}
template <class Record>
void Sortable_list<Record> :: swap(int low, int high)
/*Pre: low and high are valid positions in the Sortable_list.
  Post: The entry at position low is swapped with the entry at position high.
  Uses: The contiguous List implementation of Chapter 6.*/
{
  Record temp;
  temp=entry[low];
  entry[low]=entry[high];
  entry[high]=temp;
}
```

8.3.3 分析

＊＊＊次序不重要

对这个算法进行分析的目的，也就是关于此算法的最显著的事实是：出现的两个循环都是具有完全可预知长度的 for 循环，这意味着我们可以预先准确地计算出循环的次数。在它的比较次数中，选择排序不关心表的初始次序，因此对次序近乎正确的表，选择排序可能比插入排序慢很多。另一方面，选择排序确实具有可预知性的优点，它的最坏情况的排序时间与其最佳情况没有多少区别。

＊＊＊选择排序的优点

选择排序的主要优点与数据移动有关。如果某个元素位于正确的最终位置上，则它不会被移动。选择排序每次交换一对元素，它们当中至少有一个将被移到其最终的位置上，因此对 n 个元素的表进行排序时总共进行至多 $n-1$ 次交换。在所有的完全依靠交换去移动元素的排序方法中，这种方法属于非常好的一种。

＊＊＊分析

＊＊＊选择排序的比较次数

我们可以按函数 selection_sort 设计的同样方式去分析它的性能。主函数除了进行一些簿记和调用子程序外，不再做其他的事情。函数 swap 被调用 $n-1$ 次，其中每次调用包含 3 次元素赋值，总共是 $3(n-1)$次。函数 max_key 被调用 $n-1$ 次，其子表的长度范围从 n 下降到 2，如果 t 是 max_key 被调用的那部分表中的元素数，则为了确定最大的键，它恰好进行 $t-1$ 次键比较，因此总共有 $(n-1)+(n-2)+\cdots\cdots+1=\frac{1}{2}n(n-1)$次键比较，近似为 $\frac{1}{2}n^2+O(n)$。

8.3.4 比较

我们暂停一下，先去比较选择排序和插入排序的次数，结果为：

	选择	插入(平均)
元素的赋值	$3.0n+O(1)$	$0.25n^2+O(n)$
键的比较	$0.5n^2+O(n)$	$0.25n^2+O(n)$

从这些数字中即可以看出这两种方法的相对优点，当 n 变大时，$0.25n^2$ 远比 $3n$ 大得多。如果移动元素是一个缓慢的过程，则插入排序比选择排序所花的时间长得多。但平均来说，插入排序所进行的比较次数大约仅是选择排序的一半，如果表内的元素较小，移动起来不慢的话，则插入排序会比选择排序快。

练习 8.3

E1. 手工跟踪对下列每个表进行选择排序时的步骤。在每种情况中，计算所进行的比较次数和元素的移动次数。

(1)下列 3 个单词按字母顺序排序：

triangle square pentagon

(2)对(1)中的 3 个单词按对应的多边形的边数，以升序排序。

(3)对(1)中的 3 个单词按对应的多边形的边数，以降序排序。

(4)将下列 7 个数以升序排序：

26 33 35 29 19 12 22

(5)同样的 7 个数以不同的初始顺序，再次以升序排序：

12 19 33 26 29 35 22

(6)将下列 14 个名字按字母顺序排序：

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

* * * 计数排序

E2. 存在一种简单的算法称为计数排序(count sort)，它从未排序的表开始，在一个新数组中，构造一个新的、有序的表，假设保证原始表中的所有的键都互不相同。计数排序在表上只从头到尾做一趟，对每个记录，扫描表以计算表中有多少记录的键比它较小。如果 c 是这个计数，则这个键在有序表中的合适位置就是 c。判断计数排序要进行多少次键比较，它是否是一种比选择排序更好的算法?

程序设计项目 8.3

P1. 运行 8.2 节项目 P1 中编写的测试程序，比较选择排序和插入排序(顺序版本)，使用插入排序中用过的同样的测试数据文件。

P2. 编写并测试选择排序的链式版本。

8.4 希尔排序

如我们所看到的，插入排序和选择排序在某些方面的表现正好相反，选择排序非常有效

地移动元素，但它要进行许多冗余的比较。在最佳情况下，插入排序有最少的比较次数，但它在元素移动方面效率低下，每次只移动一个位置。我们现在的目标是导出一种方法，它尽可能地避免插入排序和选择排序的问题。我们从插入排序开始，并考虑如何能减少它移动元素的次数。

* * * 缩小增量

插入排序仅移动元素一个位置的原因在于它只与毗邻的键进行比较。如果我们准备修改它，使它首先比较远离的键，那么它就能对远离的元素排序。之后，再对较近的元素一起排序，最后被比较的键之间的增量将降到1，保证这个表是完全有序的。这是1959年D. L. SHELL在以他自己的名字命名的排序方法中实现的思想，这一思想有时也称为缩小增量排序(diminishing-increment sort)。在正式描述这个算法之前，我们先来看一个对名字进行排序的例子。

* * * 例子

图8.7说明当我们首先以彼此相隔5个距离对所有名字进行排序时发生的情况(因此在每个这样的表中将仅有两、三个名字)，然后用增量3重新对名字排序，最后执行普通的插入排序(增量1)。

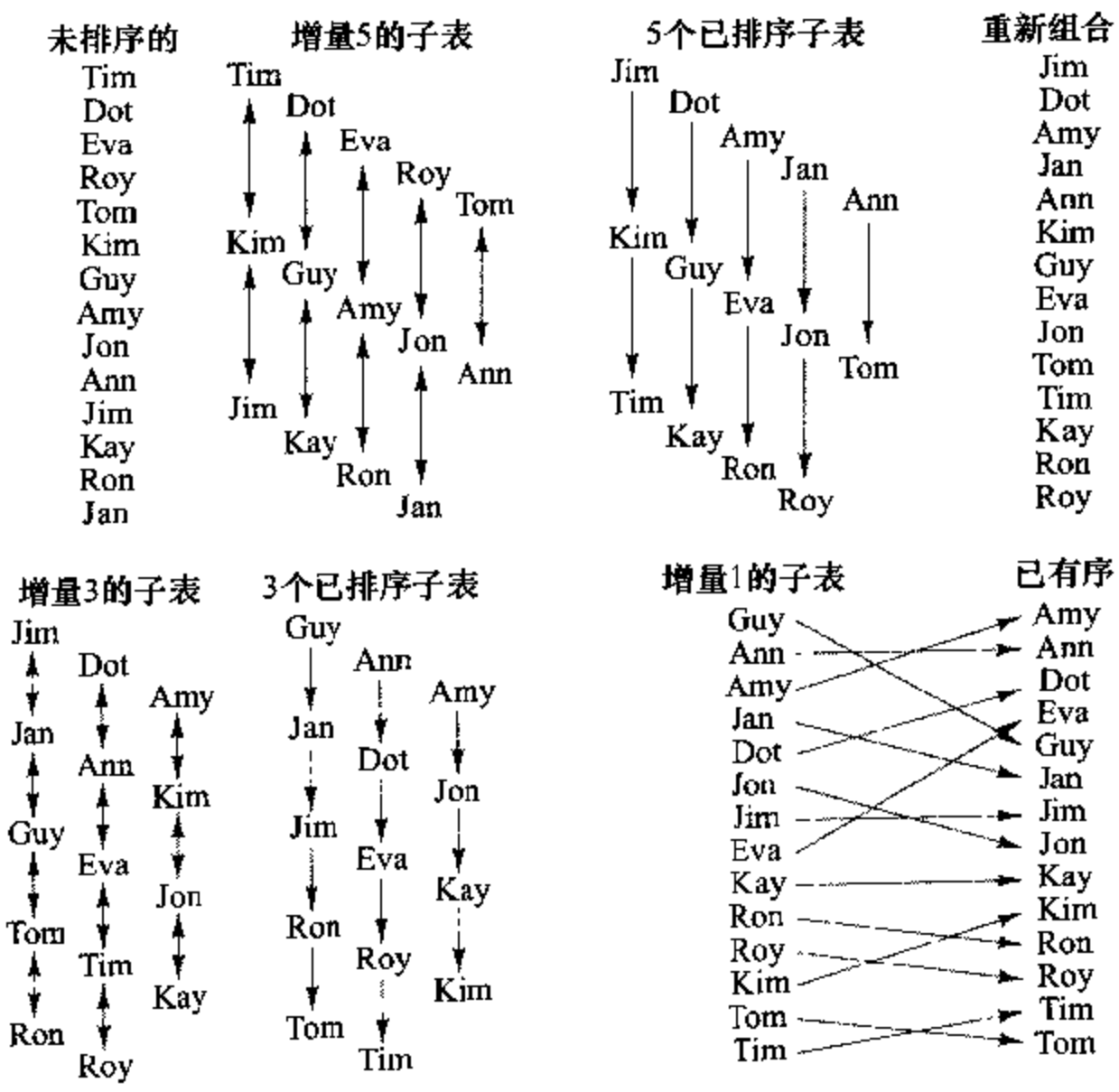


图8.7 希尔排序的例子

你能注意到，即使在所有名字上做三趟排序，前面两趟已将各名字移到其最终位置的附近，因此最后一趟排序(它做普通的插入排序)中，所有的元素与其最终的位置都很近，因此这趟排序很快。

* * * 增量的选择

选择5,3和1作为增量并不神奇,其他的一些选择可能效果也一样好或者会更好。然而,选择2的幂如8,4,2和1则可能比较浪费,因为在某一趟比较过的同样的键会在下一趟被再次比较,因此通过选择互相不是倍数的数字,更有机会从比较中得到更多的新信息。虽然对希尔排序有过一些研究,但尚未能够证明一种增量选择远比所有其他的选择更优。对增量的选择也有各种建议,如果选择的增量太靠近,就像我们刚才那样,那么就需要更多的趟数,但每一趟可能更快些。如果增量下降太快,则需要较少的趟数,但每一趟会较长。它的惟一本质的特征是最后的增量为1,因此在此过程结束之时,对表进行检查以保证它是完全有序的。为后继算法的简单性,我们从增量 increment==count 开始,在6.2.2节中我们让 count 表示被排序的表 List 的长度,并且在每一趟用下面的语句去减少增量:

increment=increment /3+1;

因此算法中使用的增量与图8.7中所用的增量并不一样。

我们现在可以列出用于顺序表的算法。

```
template <class Record>
void Sortable_list<Record>::shell_sort()
/* Post: The entries of the Sortable_list have been rearranged so that the keys in all
        the entries are sorted into nondecreasing order.
  Uses: sort_interval*/
{
  int increment,              //  spacing of entries in sublist
    start;                    //  starting point of sublist
  increment=count;
  do {
    increment=increment/3+1;
    for (start=0;start <increment;start++)
      sort_interval(start,increment);     //  modified insertion sort
  } while (increment>1);
}
```

辅助成员函数 sort_interval(int start,int increment)正是函数 insertion_sort,除了表从变量 start 而不是从0开始并且连续值之间的增量是给出的值而不是1。将 insertion_sort 修改成 sort_interval 的细节留作练习。

* * * 优化的插入排序

由于希尔排序的最后一趟增量为1,实际上希尔排序是预处理阶段优化后的插入排序,它首先用较大的增量对子表进行排序以达到优化。因此关于希尔排序正确工作的证明与插入排序正确工作的证明完全一样,并且,虽然我们有较好的理由认为此预处理阶段通过消除元素的多次移动到仅移动一个位置,从而显著地加速排序,但实际上我们并未证明希尔排序在任何时候都比插入排序快。

* * * 分析

对希尔排序的分析显得非常困难,到此为止,我们仅得到特殊条件下的比较和移动次数的较好估计。了解这些数字如何依赖于增量的选择会很有趣,依此就可以做出最好的选择。但即便缺乏完整的数学分析,在计算机上运行一些较大的例子仍会令人相信希尔排序确实相当不错。对希尔排序大量的经验研究表明,当 n 较大时,它的移动次数在 $n^{1.25}$ 到 $1.6\ n^{1.25}$ 之间,这对插入排序确实是真正的改进。

练习 8.4

E1. 使用增量为(1)8,4,2,1 和(2)7,3,1 的希尔排序,手工对图 8.7 中"未排序的"栏中的 14 个名字的表进行排序,计算每种情况中所进行的比较和移动次数。

E2. 解释为何希尔排序不适合链表使用。

程序设计项目 8.4

P1. 重新编写方法 insertion_sort,让它充当 shell_sort 中嵌入的函数 sort_interval。

P2. 用 8.2 节项目 P1 的程序测试 shell_sort,使用与插入排序同样的数据文件,并比较结果。

8.5 下限

既然现在已发现有一种方法比我们介绍的第一种方法运行得更好,那么是否适于问一句:"排序可能有多快?"

为了回答这个问题,我们将注意力局限于(像我们在回答查找中的同一问题时所做的那样)完全依赖键对之间比较的排序方法。

* * * 比较树

我们在这类排序算法中任取一个,并考虑它如何对 n 个元素的表进行排序。想像地画出它的比较树。图 8.8 中显示了应用于 3 个数 a,b,c 的插入排序和选择排序的样例比较树。所进行的每一次键比较,对应于一个内部顶点(画作一个圆圈),树叶(方格结点)显示了排序

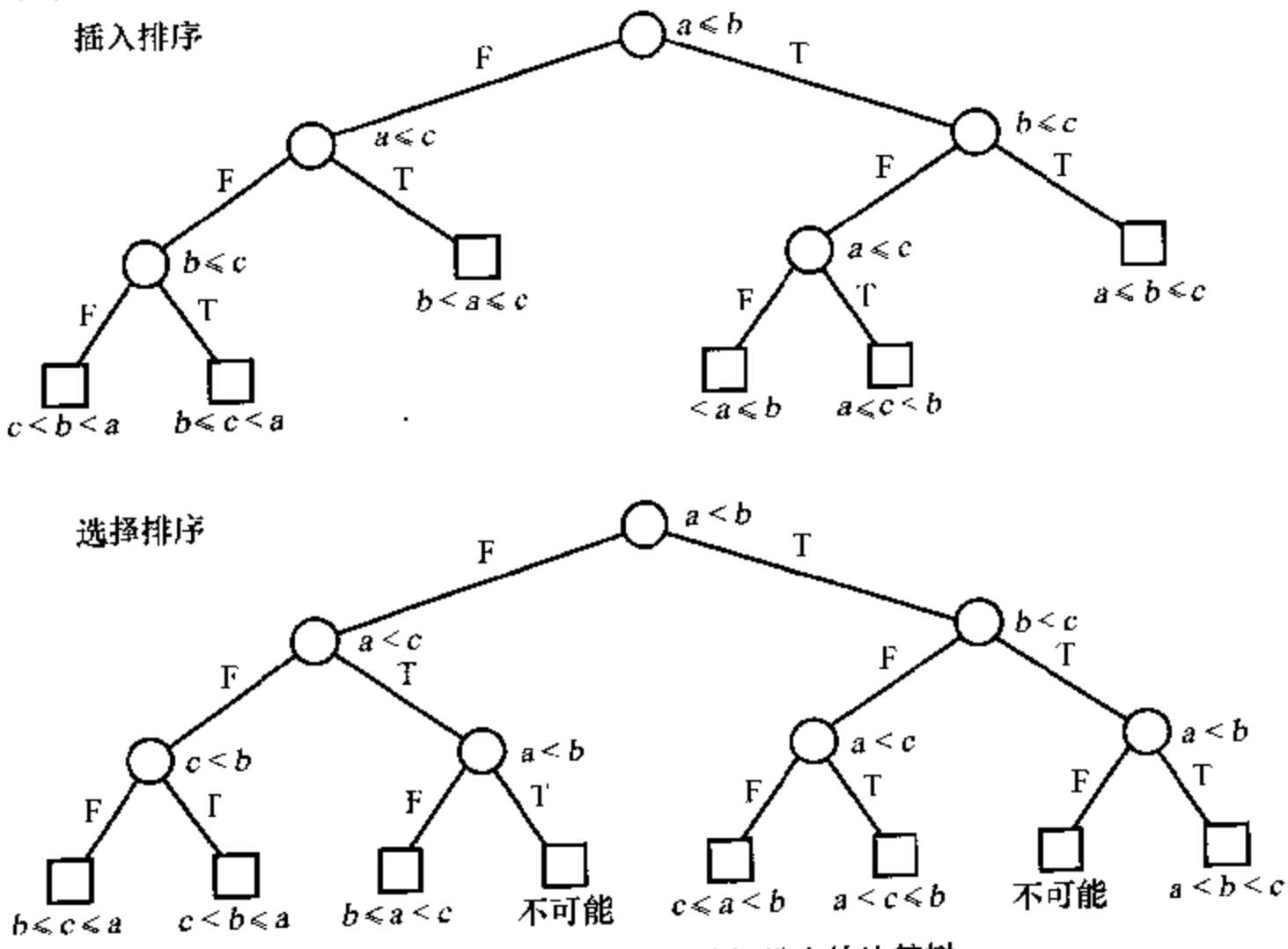


图 8.8 n=3 时插入排序和选择排序的比较树

后数字的次序。

这个图清楚地说明：平均来说，选择排序比插入排序要进行更多的键比较。实际上，选择排序要进行冗余的比较，要重复以前已进行过的比较。

＊＊＊比较树：高度和路径长度

任一种排序算法的比较树展示了这种算法的一些特性，它的高度是要进行比较的最大次数，因而给出了这个算法在最坏情况下的性能。外部路径长度在除以树叶的数目后，给出了这个算法的平均比较次数。比较树利用从根到树叶的不同路径，展示了全部可能的比较序列，由于这些比较控制着排序时元素如何进行重排，因此表的任何两种不同的次序必然导致某些不同的决定，并因此导致贯通全树的不同路径，这些路径必须以不同的树叶终止。包含 n 个元素的表最初能被排序的方法有 $n!$ 种(见 A.3.1 节)，因此树中的树叶数必须至少是 $n!$。引理 7.5 隐含地说明这棵树的高度至少是$\lceil \lg n! \rceil$，它的外部路径长度至少是$n!\ \lg n!$。(回忆一下，$\lceil k \rceil$表示不小于 k 的最小整数。)将这些结果转换成比较次数，我们可以得到：

定理 8.2 利用键比较对 n 个元素的表进行排序的任何算法，在最坏情况下必须进行至少$\lceil \lg n! \rceil$次键比较，在平均情况下，必须进行至少 $\lg n!$ 次键比较。

＊＊＊对 $\lg n!$ 取近似

Stirling 公式(附录 A 中定理 A.5)给出一个整数的阶乘的近似值，$n!$ 在取以 2 为底数的对数后，为：

$$\lg n! \approx (n+\frac{1}{2})\lg n-(\lg e)n+\lg\sqrt{2\pi}+\frac{\lg e}{12n}$$

这个表达式中的常数有如下近似值：

$\lg e \approx 1.442695041$ 和 $\lg\sqrt{2\pi} \approx 1.325748069$

$\lg n!$ 与其 Stirling 近似实际上非常接近，已超过我们分析算法所需要的接近程度。对几乎所有的目的，下面这个粗略近似证明会非常令人满意：

$$\lg n! \approx (n+\frac{1}{2})(\lg n-1\frac{1}{2})+2$$

我们经常仅使用近似值 $\lg n! = n\lg n+O(n)$。

＊＊＊其他方法

在结束本节之前，我们特别指出有时排序方法不用比较而可能更快。例如如果事先知道有 100 个元素，并且它们的键正好是某种次序的 1 到 100 之间的整数，没有重复的值，那么对它们进行排序的最好办法是不进行任何键比较，而只是看某个元素有键 i 的话，就将它放到位置 i。用这种方法，我们(至少暂时地)认为这些元素是存储在一个表格而不是一个表中，随后就能用每个元素的键作为下标去找到它在表格中的合适位置。项目 P1 扩展这种思想，使其成为一种算法。

练习 8.5

E1. 画出用于 4 个对象的(1)插入排序和(2)选择排序的比较树。

E2. (1)找出一种用于 4 个键的排序方法，它在最坏情况下有最少可能的键比较次数，因而

在这方面是最优的。(2)找出你的算法在平均情况下进行多少次比较(应用于 4 个键)。修改你的算法,使它尽可能接近达到 lg 4!≈4.585 次键比较的较低下限,为什么它不可能达到这个较低下限?

E3. 假设有一副 52 张洗过的纸牌,共有 4 种花色,每种花色有 13 张牌。现在希望对这副牌进行排序,使得 4 种花色是有序的,并且每种花色中的 13 张牌也是有序的。下列方法中哪一种最快?

(1)从头到尾查看这副牌,从中移出梅花,然后对它们单独排序。继续对方块、红心和黑桃使用相同的过程。

(2)按牌的大小将这些牌分成 13 堆,将这 13 堆又重新堆回到一起,然后按花色分成 4 堆,再将这 4 堆又堆回到一起。

(3)通过将每张牌放在相对于前面已有序的牌的合适位置上,只对这些牌从头到尾做一趟排序。

程序设计项目 8.5

本节的排序项目是要求某种特殊类型的键的专用方法,这些键是从 0 到 1 之间的伪随机数,因此它们不适用于本章中为研究其他方法而设计的测试程序,也不使用与其他方法同样的数据。

* * * 插值排序

P1. 构造一个由 n 个严格在 0 到 1 之间的伪随机数组成的表,合适的 n 值为 10(用于调试)和 500(用于与其他方法比较结果)。编写一个程序通过下面的插值排序(interpolation sort)在一个数组中对这些数进行排序。首先,清除此数组(全部为 0),对旧表中的每个数,将它乘上 n 并取整数部分,然后查找表格中的那个位置,如果那个位置是 0,则将此数放在那里,如果不是 0,则(按当前数与已在此位置上的那个元素的相对大小)左移或右移,以找到插入新数的位置。如果需要腾出空间,则移动表中的元素(按插入排序的方式)。证明你的算法确实能正确地对这些数进行排序,并将它们的运行时间与应用于同样长度的随机有序表上的其他排序方法进行比较。

* * * 链分配排序

P2. (由 B. Lee 提出)编写一个程序执行下面的链分配排序。同上个项目一样,假设键是在 0 到 1 之间的伪随机数。创建一个数组链表,并按大小将这些键分配到链表中。这些链表或者在插入数时保持有序,或者在第二趟将这些表一起链成一个有序表的过程中再进行排序。通过试验确定要使用的表的最优数目。(看起来当有足够的表使得每个表的平均长度大约是 3 的时候效果较好。)

8.6 分而治之排序

8.6.1 主要思想

＊＊＊越短越简单

重新开始通常是一个好办法，我们要（暂时）忘掉所知道的有关排序的所有知识，仅努力应用一条在先前研究的算法中呈现出来并由一般经验而知的重要原理：对短表的排序比长表更简单。如果待排序的表的长度翻倍，则所做的工作就不止翻倍（粗略地看，对插入排序或选择排序来说是4倍）。因此如果能找到一种办法，将表分成长度大致相等的两个表并对它们分别排序，则可以节省工作。例如，如果你在某个图书馆工作，现在给你1000张索引卡片，要求按字母顺序摆放，那么一种好办法是将它们按首字母分成堆，然后分别对各堆排序。

＊＊＊分而治之

这里我们再次应用将一个问题分成更小的但类似的子问题的思想，即分而治之的思想。

首先，我们指出，计算机进行的比较一般是两路分支，因此在这个过程的每个阶段，我们将元素分在两个表中进行排序。

有人可能会问，应使用什么方法对缩短后的表进行排序？由于我们（暂时）已忘记了自己所知道的所有其他方法，因此我们再次使用同样的方法，即分而治之方法去重复地再分这个表。但即使我们不知道正式的排序方法，也不用一直做下去，因为对只有一个元素的表进行排序时不用做任何工作。

总而言之，让我们非正式地列出分而治之排序：

```
void Sortable_list::sort()
{
  if the list has length greater than 1 {
    partition the list into lowlist,highlist;
    lowlist.sort();
    highlist.sort();
    combine(lowlist,highlist);
  }
}
```

我们仍然必须确定如何将这个表分成两个子表，以及在排序后如何将子表合成为单个的表。有两种方法，每一种方法在不同的环境中效果很好。

＊＊＊归并排序

- 归并排序（Mergesort）：在第一种方法中，我们简单地将表分成长度尽可能相等的两个子表，并随后对它们排序。在排序后，又小心地将两个有序的子表归并成一个单一的有序表，因此这种方法称为归并排序。

＊＊＊快速排序

＊＊＊支点

- 快速排序（Quicksort）：第二种方法在第一步将表划分成两个表的时候已经做了大部分的工作，因而最后一步合成子表就变得简单了。这种方法是由C. A. R. HOARE发明并命名为快速排序。为了划分一个表，首先从表中选择某个键，希望这个键的前后部

分各有大约一半的键，我们称这个选定的键为支点(pivot)。在接下来对元素的划分中，使键比支点小的所有元素在一个子表中，键比支点大的元素分到另一个子表中。最后，对这两个缩短后的表分别排序，再将它们放在一起，使得整个表是有序的。

8.6.2 例子

在将我们的方法细化成详细的函数前，先来完成一个具体的例子，对下面 7 个数排序：

26 33 35 29 19 12 22

1. 归并排序的例子

* * * 约定：左表较长

第一步是将表分成两个子表，当(如此例中)表的长度为奇数时，我们建立一个约定：让左边的子表比右边的子表多一个元素。因而将上表分成：

26 33 35 29 和 19 12 22

* * * 第一半

首先考虑左边的子表，再次将它分为两半，变成

26 33 和 35 29

对其中的每个子表，再次应用同样的方法，将它们中的每一个再分成均只有一个数的子表。长度为 1 的子表当然不要求排序。最后，就能开始归并这些子表以得到一个有序表。子表 26 和 33 归并成有序表 26 33，子表 35 和 29 归并成 29 35。下一步，归并这两个长度为 2 的有序子表，得到一个长度为 4 的有序子表 26 29 33 35。

* * * 第二半

现在原始表的左半部分已是有序的，我们对右半部分执行同样的步骤。首先将它分成子表 19 12 和 22。其中的第一个子表要分为两个长度为 1 的子表，再进行归并后得到 12 19。第二个子表 22 的长度为 1，故不需排序，现在将它与 12 19 归并成有序表 12 19 22。

最后，长度为 4 和 3 的有序子表被归并成：12 19 22 26 29 33 35。

图 8.9 中画出的归并排序的递归树显示了所有这些子表和递归调用，递归调用发生的

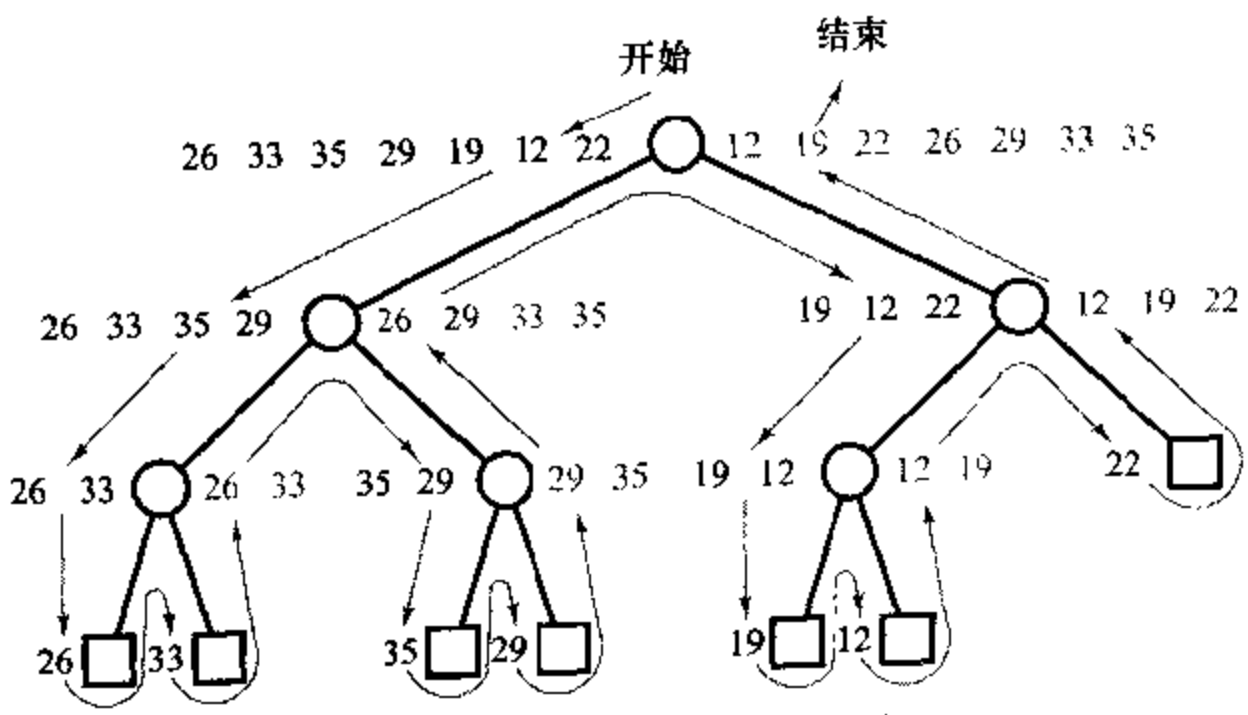


图 8.9 7 个数归并排序的递归树

次序用浅灰色路径显示，传递给递归调用的每个子表中的数字以黑色显示，在归并后有次序的数字以浅灰色显示。不需进一步递归的调用(长度为 1 的子表)是树的树叶并画作方格。

2. 快速排序的例子

＊＊＊支点的选择

我们仍然用同样的例子，这次是运用快速排序，仔细记下我们的概要方法生成的运行步骤。为使用快速排序，首先必须确定选择哪个键作为支点以将表分成两半。可以自由地选择任何数，但为一致性起见，我们将采用一条确定的规则。或许最简单的规则就是选择表中的第一个数作为支点，而且我们在这个例子中也是这样做的。然而，对实际应用来说，其他的选择通常比第一个数更好。

＊＊＊划分

我们的第一个支点是 26，划分得到的两个子表是：19 12 22 和 33 35 29，它们分别包含小于和大于支点的数。我们保持子表中元素的次序与其在原始表中的次序不变，但这种决定也是任意的。某些版本的快速排序将支点放入其中一个子表中，但我们选择不将支点放在任一个子表中。

现在就到了概要的下一行，它告诉我们对第一个子表进行排序，因此我们从头开始再次启动这个算法，但这一次是用于较短的表 19 12 22。

＊＊＊低半部分

这个表的支点是 19，它将这个表分成两个分别只有一个数的子表，其中第一个子表是 12，第二个子表是 22。由于只有一个元素，因此这两个子表都不用排序，因此就到了概要的最后一行，在这里将两个子表合并并将支点放在二者中间，合成为有序的表 12 19 22。

现在对这个子表的 sort 函数的调用已经结束，故它返回被调用的位置，它是在七个数的完全表的 sort 函数中被调用的，因此现在回到那个函数的下一行。

＊＊＊内部和外部函数调用

我们现在两次调用这个函数，其中第二个实例是发生在第一个实例的内部。仔细注意此函数的两个实例是在不同的表上工作，且不同于在一个循环内执行同样的代码两次。将这两个实例看作具有不同的颜色可能有所帮助，使得能够以不同的颜色全部写出第二个(内部)调用中的命令来代替这次调用，从而清晰地将它们作为单独的函数实例区分开来。图 8.10 中说明了这个过程的步骤。

返回到这个例子，我们发现函数的第一个实例的下一行是另一个调用，它对另一个表排序，这次是 3 个数 33 35 29。

＊＊＊高半部分

同前面的(内部)调用一样，支点 33 直接划分这两个表，得到长度为 1 的两个子表，并随后合并它们生成有序表：29 33 35。

最后，对 sort 的这个调用返回，我们到达对完全表排序的(外部)实例的最后一行，此时，长度为 3 的两个子表与原来的支点一道合并为有序表 12 19 22 26 29 33 35。

＊＊＊重新合成

在此步骤之后，这个过程结束。

记录快速排序例子所有调用的一种简单办法是画出它的递归树，如图 8.11 中所示。其

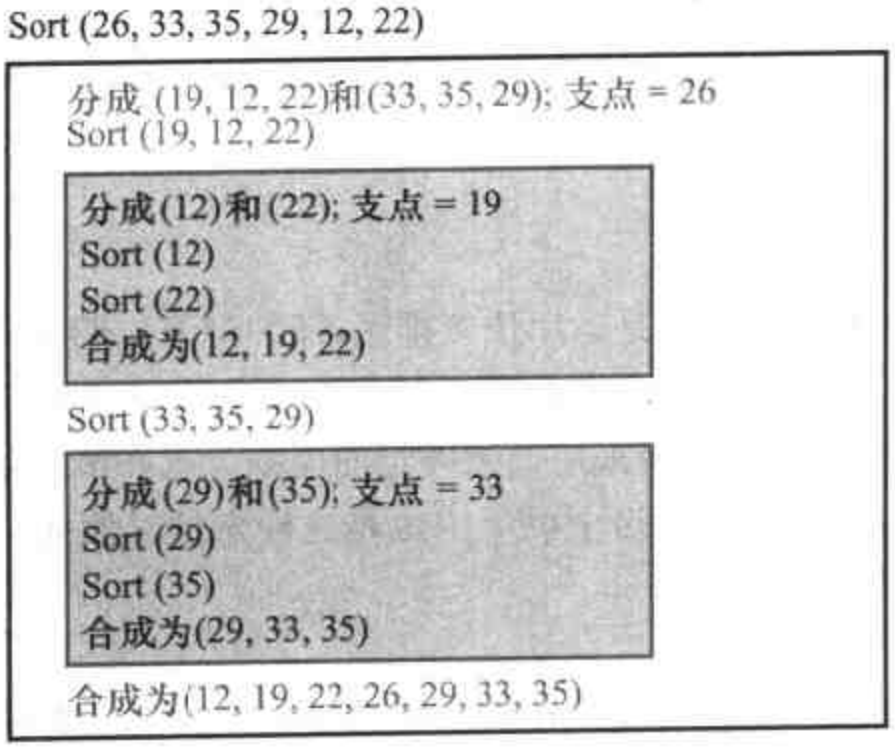


图 8.10 快速排序的执行踪迹

中每一层对 sort 的两次调用显示成此顶点的孩子，长度为 1 或 0 的子表不需排序，因此画成树叶。在其他的顶点上(为避免弄乱这个图)，我们只包含用作调用的支点。然而不难读出每个子表中的所有数(但不必按它们的初始次序)。每个递归调用的子表中的数是对应顶点上的数和这个顶点的所有后代上的数。

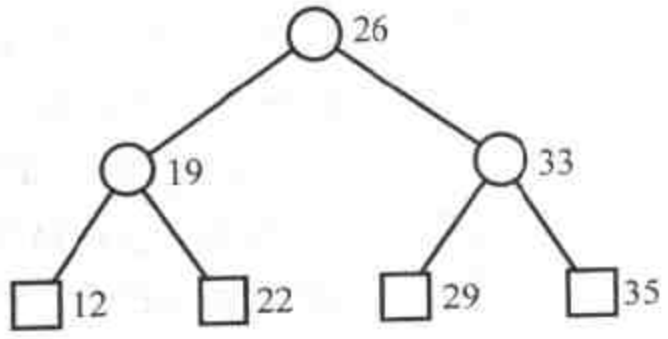


图 8.11 7 个数的快速排序的递归树

* * * 例子

如果你对递归的使用仍感到困难，那么你就应该先暂停一下，做完前一节中介绍的对 14 个名字的表使用归并排序和快速排序方法进行的排序。作为一个检查，图 8.12 使用与前一个名字的表使用归并排序和快速排序方法进行的排序。作为一个检查，图 8.12 使用与前一例子中使用的同样简略的形式给出了快速排序的调用树，这棵树以两种版本给出，其中一种版本中用每个表的第一个键作为支点，另一个表使用中央的键(对偶数长度的表为中央偏左的键)作为支点。

练习 8.6

E1. 将快速排序应用于本节中讨论的 7 个数的表，每个子表中的支点选择为：(1)子表的最后一个数；(2)子表的中央(或左中央)的数。在每种情况中，画出递归调用树。

E2. 将归并排序应用于先前排序方法中考虑的 14 个名字的表：

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

E3. 将快速排序应用于 14 个名字的这个表，并手工将它们按字母顺序排序。将支点取为

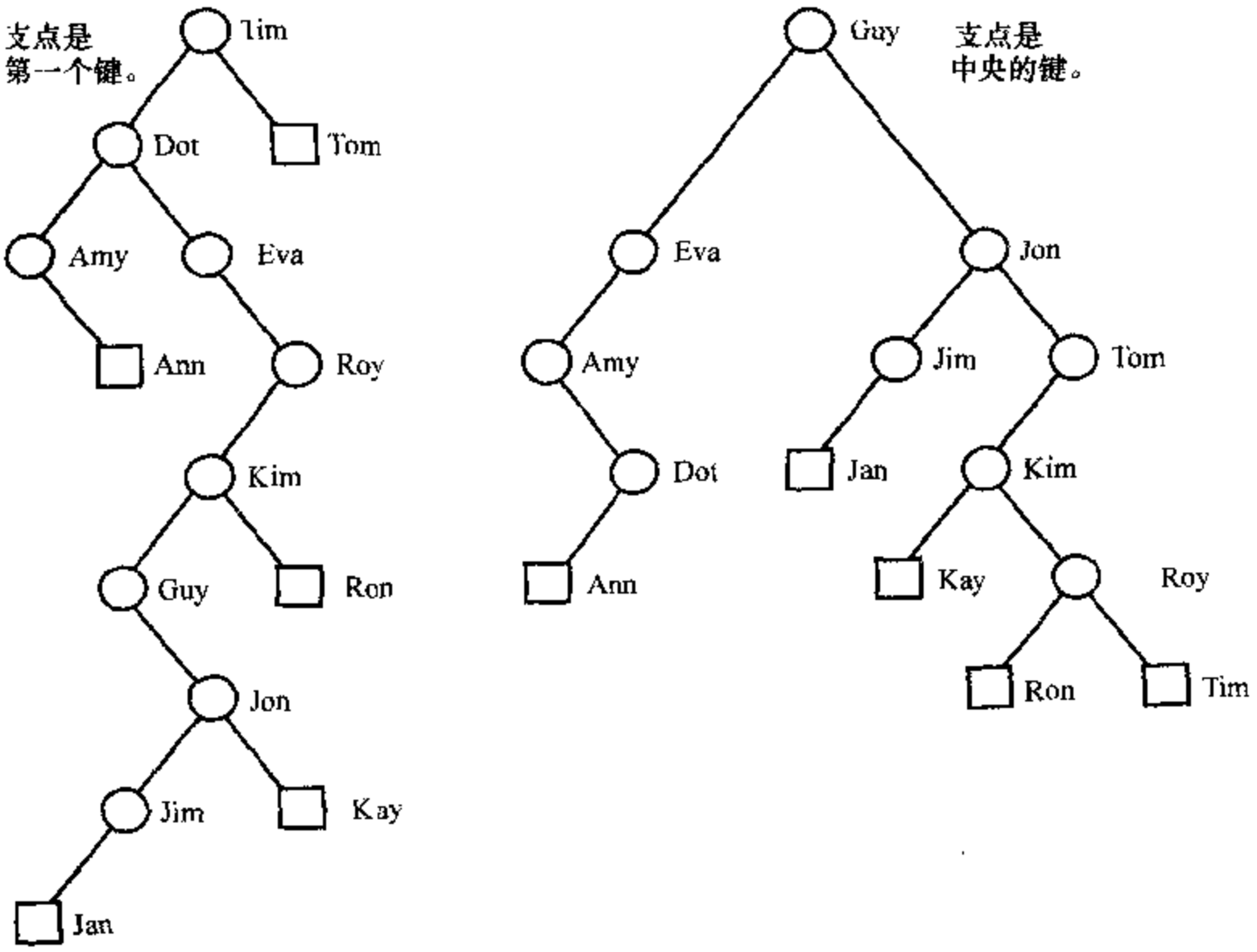


图 8.12　14 个名字的快速排序的递归树

(1)每个子表中的第一个键(2)每个子表的中央键(或左中央)，见图 8.12。

E4. 在两种分而治之方法中，我们试图将一个表分成长度近似相等的子表，但即使不是相等长度的两半，分而治之排序的基本框架仍是合理的。考虑分割一个表，使得其中一个子表的长度为 1。可根据是从表中分出一个元素时还是在子表合并时完成工作，得到两种方法。

(1)通过找出具有最大键的元素并将它作为长度为 1 的子表，以此分出这个表。在对剩余元素排序后，通过将最大键的元素放在最后而轻易地合并子表。

(2)将表的最后一个元素分出来，在对剩余元素排序后，将这个元素归并到表中。

证明其中的一种方法与插入排序相同，而另一种方法与选择排序相同。

8.7　链表的归并排序

我们现在开始为分而治之排序的每种方法编写正式的函数。就归并排序来说，我们要编写一个用于链表的版本，并将顺序表的版本留作一个练习。对快速排序，我们反过来，只为顺序表编写代码。但这两种方法都能适用于顺序表和链表。

归并排序也是一种极好的外部排序方法，即针对数据保存在磁盘上而不是高速内存中的问题。

8.7.1 函数

对链表排序时，我们重排这个表的链并避免创建和删除新的结点。特别地，归并排序必须调用一个递归函数，此函数处理正在排序的表结点的子集，我们称这个函数为 recursive_merge_sort。归并排序的主要实现仅是在对 recursive_merge_sort 的调用中将一个指针传递给表的第一个结点。

* * * 主函数

```
template <class Record>
void Sortable_list<Record>::merge_sort()
/*Post: The entries of the sortable list have been rearranged so that their keys are
        sorted into nondecreasing order.
  Uses: The linked List implementation of Chapter 6 and recursive_merge_sort.*/
{
  recursive merge_sort(head);
}
```

归并排序的基本方法的框架直接转换成下面的递归排序函数：

```
template <class Record >
void Sortable_list<Record >::recursive_merge_sort(Node<Record>*&sub_list)
/*Post:The nodes referenced by sub_list have been rearranged so that their keys are
       sorted into nondecreasing order. The pointer parameter sub_list is reset to
       point at the node containing the smallest key.
  Uses:The linked List implementation of Chapter 6; the functions divide_from, merge,
       and recursive_merge_sort.*/
{
  if (sub_list!=NULL && sub_list->next!=NULL){
     Node<Record>* second_half=divide_from(sub_list);
     recursive_merge_sort(sub_list);
     recursive_merge_sort(second_half);
     sub_list=merge(sub_list,second_half);
  }
}
```

函数 recursive_merge_sort 中的参数 sub_list 是一个指向结点的指针的引用，需要这个引用去允许函数修改调用自变量。

recursive_merge_sort 调用的第一个辅助函数是：divide_from(Node<Record> * sub_list);。它采用由参数 sub_list 引用的表，并通过用 NULL 指针替代其中间的链而将这个表分成两半，此函数返回指向原始子表的第二部分的第一个结点的指针。

* * * 将一个链表分成两半

```
template <class Record>
Node<Record>* Sortable_list<Record>::divide_from(Node<Record>*sub_list)
/*Post: The list of nodes referenced by sub_list has been reduced to its first half, and
        a pointer to the first node in the second half of the sublist is returned. If the
        sublist has an odd number of entries, then its first half will be one entry
        larger than its second.
  Uses: The linked List implementation of Chapter 6.*/
 {
```

```
Node<Record>* position,       //  traverses the entire list
             * midpoint,      //  moves at half speed ofposition to midpoint
             * second_half;
if ((midpoint=sub_list)==NULL) return NULL;     //  List is empty.
position=midpoint->next;
while (position!=NULL){//  Move position twice for midpoint's one move.
  position=position->next;
  if(position!=NULL){
    midpoint=midpoint->next;
    position=position->next;
  }
}
second_half=midpoint->next;
midpoint->next=NULL;
return second_half;
}
```

* * * 归并中的极端情况

第二个辅助函数是：Node<Record> * merge(Node<Record> * first,Node<Record> * second);。这个函数归并由 first 和 second 引用的结点的表，返回指向具有最小键的归并表的结点的指针。这个函数中的大部分工作是比较一对键，其中每个键各来自一个表，并将适当的那一个附加到归并表中。然而在表的起始和结尾处都要特别小心。在结尾处，表的 first 和 second 之一会在另一个之前被用尽，这时仅需要将剩余链表的其他部分附加到归并表中即可；而在起始处，必须用一个指针记住归并表的第一个结点，此结点将作为函数值返回。

* * * 哑结点

为记录归并表的起始而不必考虑一些特殊情况，我们在 merge 函数中声明了一个称为 combined 的临时 Node，并在查看任何实际的键之前，将它放在归并表的起始处（即强迫归并表从已在其中的一个结点开始），然后可以将实际的结点插入而不必考虑特殊情况。在结束时，combined 将包含一个指向归并表的首个实际结点的指针，因此函数能够返回这个指针。临时结点 combined 称为哑结点，因为它不包含实际的数据，而仅用作简化指针操作。

图 8.13 中说明了函数 merge 的操作。

* * * 归并两个已排序的链表

```
template <class Record>
Node<Record>*Sortable_list<Record>::merge(Node<Record>*first,
                                          Node<Record>*second)

/*Pre: first and second point to ordered lists of nodes.
  Post:A pointer to an ordered list of nodes is returned. The ordered list contains all
       entries that were referenced by first and second. The original lists of nodes
       referenced by first and second are no longer available.
  Uses:Methods for Record class;the linked List implementation of Chapter 6.*/
{
  Node<Record>*last_sorted;     //  points to the last node of sorted list
  Node<Record>combined;         //  dummy first node, points to merged list
  last_sorted=&combined;
  while (first!=NULL && second!=NULL){   //  Attach node with smaller key
```

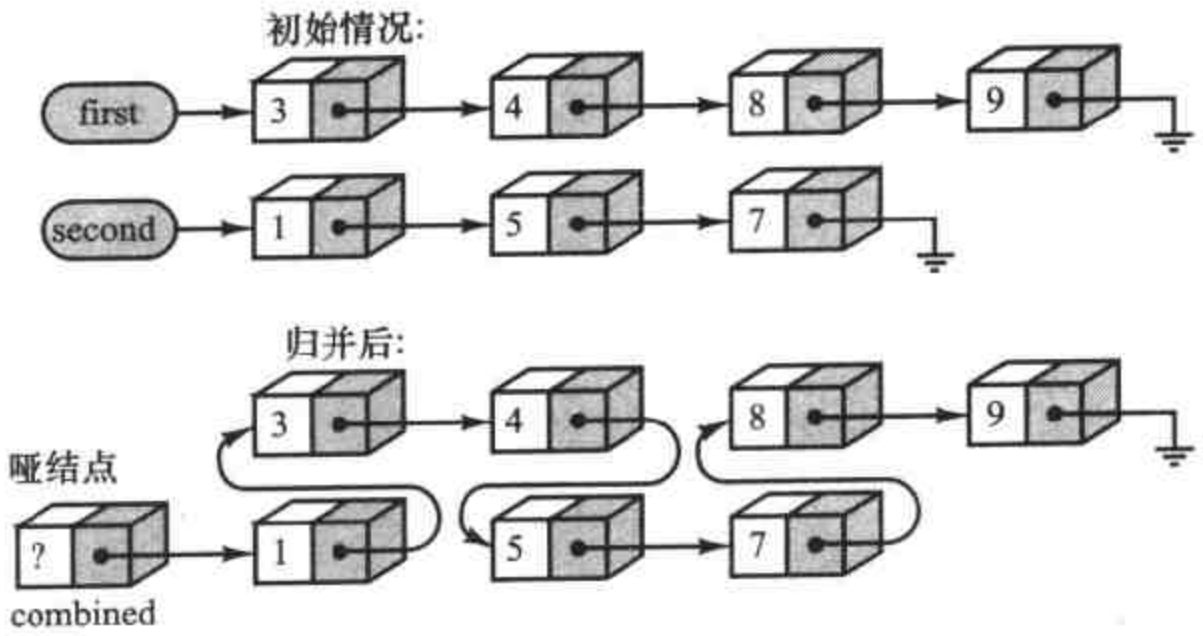


图 8.13 归并两个已排序的链表

```
if(first->entry <=second->entry) {
   last_sorted->next=first;
   last_sorted=first;
   first=first->next;   //  Advance to the next unmerged node.
}
else {
   last_sorted->next=second;
   last_sorted=second;
   second=second->next;
}
//  After one list ends, attach the remainder of the other.
  if (first==NULL)
     last_sorted->next=second;
  else
     last_sorted->next=first;
  return combined.next;
}
```

8.7.2 关于归并排序的分析

既然我们有了一个能运行的归并排序函数,现在可暂停一下,先去确定一下它的性能,以便能与其他的排序方法进行合理的比较。与链表上的其他排序算法一样,我们不必关心移动元素所需的时间,因而我们将注意力放在函数要进行的键比较次数上。

1. 比较次数

* * * merge 函数

在整个归并排序函数中,只在一个位置进行键比较,这个位置是在 merge 函数的主循环中。在每次比较后,两个结点中的一个被送到输出表,因此比较次数必定不会超过被归并的元素个数。为找出这些表的总长度,我们再次考虑此算法的递归树,为简单起见,我们在图 8.14 中给出了 $n=2^m$ 为 2 的幂的情况。

图 8.14 的树很明显,每一层上的表的总长度正好是 n,即元素的总数。换言之,在每一

层上，每个元素正好由一个归并处理，因此每一层上的比较总数不能超过 n。除树叶所在的层(它不进行归并)以外，层数是将 $\lg n$ 上舍入到下一个最小整数，即上限$\lceil \lg n \rceil$。因而，在 n 个元素的表上，归并排序的键比较次数不超过$\lceil n \lg n \rceil$。

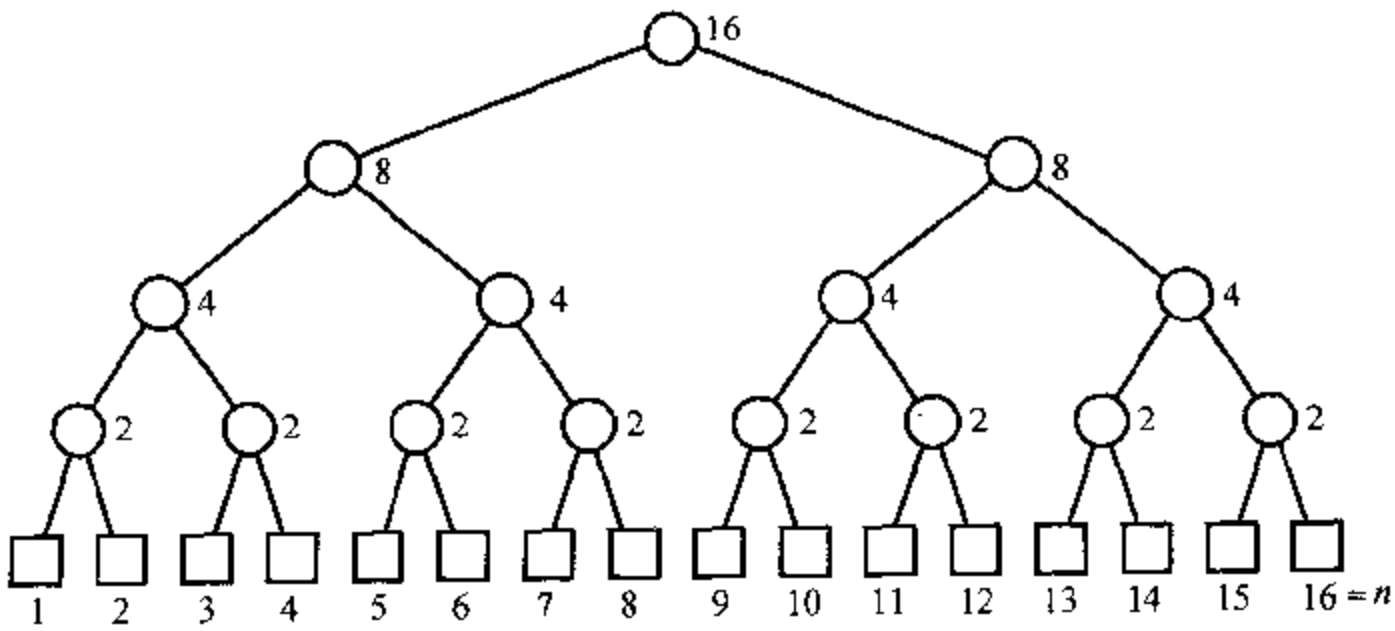


图 8.14　子表归并的长度

2. 与插入排序对比

回忆一下(8.2.4 节)，平均来说，插入排序在对 n 个元素排序时要进行超过$\frac{1}{4}n^2$ 次键的比较。一旦 n 超过 16，$\lg n$ 就比$\frac{1}{4}n$ 小。当 n 是排序一个表的实际长度时，$\lg n$ 远比$\frac{1}{4}n$ 小，因此归并排序的比较次数远小于插入排序的比较次数。例如当 $n=1024$ 时，$\lg n=10$，因此归并排序的比较限度是 10 240，而插入排序的平均比较次数超过 250 000。同一问题，如果使用插入排序需要用一分钟计算机时间，那么使用归并排序时可能只需要一或两秒钟。

* * * $n \lg n$

在前面的计算中表达式 $n \lg n$ 的出现决不是偶然的，但它与 8.5 节中建立的下限密切相关，在那里我们证明任何使用键比较的排序方法必须至少进行 $\lg n! \approx n\lg n - 1.44n + O(\log n)$次键比较。当 n 较大时，这个表达式中的第一项变得比其他项更重要。我们现在已经找到了达到此下限范围内的算法，即归并排序。

3. 改进次数

实际上，如果稍微再仔细一点，我们能得到归并排序进行比较的更精确的计数，它表明归并排序的实际性能甚至更接近于下限所允许的最佳可能的键比较次数。

首先，我们注意到归并合计长度为 k 的两个表不会需要 k 次比较，而至多是 $k-1$ 次，因为在输出次大的键以后，就没有留下的键要与最大的键比较了，故不需另一次比较就可以输出最大的键。所以对执行的每趟归并，我们能将总共的比较次数减 1。总的归并实质上是：

$$\frac{n}{2}+\frac{n}{4}+\frac{n}{8}+\cdots+1=n-1$$

(当 n 为 2 的幂时,这个计算是准确的,其他情况下是一种较好的近似。)归并排序所进行的键比较的总次数因此小于 $n \lg n - n + 1$。

*** 改进的次数

其次,我们要指出,正被归并的两个表中的一个有可能在另一个之前被归并完,此时第二个表中的元素不用更多的比较即可直接取出,因此比较次数还要小于我们刚才的计算。例如一个表中的每个元素可能先于第二个表的每个元素,因此第二个表中所有元素都不用比较即可输出。练习中略述了总的次数平均能减少到 $n \lg n - 1.1583n + 1$ 的证明,n 的正确系数可能接近于 -1.25。我们因此发现,不仅主项与允许的下限一样小,同时第二项也十分接近。通过进一步细化 merge 函数,这个方法能被纳入与理论上最优的比较次数相差百分之几的范围之内(见参考书目)。

4. 结论

*** 链式归并排序的优点

从这些评论看来,归并排序是最终的排序方法,实际上,对随机次序的链表而言,确实难以超越它。然而我们必须记住,除了键比较以外还有其他的考虑也很重要,我们所编写的程序要花费许多时间去查找表的中央元素以将表分成两半。练习中讨论了一种节省这个时间的方法。链式版本的归并排序能有效地使用空间,它不需要大的辅助数组或其他的表,并且由于递归的深度仅为 $\lg n$,因此用于跟踪递归调用所需的空间量也非常少。

5. 顺序归并排序

*** 归并的三方面权衡

遗憾的是,对顺序表来说,归并排序并不是这样绝对的成功,其困难在于归并两个顺序表而不显著花费下列 3 个方面的代价:

- 空间;
- 计算机时间;
- 程序设计工作。

归并两个顺序表的第一种办法,并且也是最直接的办法是使用一个大得足够容纳合并表的辅助数组,然后在对表进行归并时将元素复制到这个数组中。这种方法要求 $\Theta(n)$ 的额外空间。在第二种方法中,我们将待归并的两个表互相接起来,忘掉它们已被排过序,然后使用像插入排序这样的方法对合并后的表进行排序。这种方法几乎不使用额外空间,但是同与 n 成比例的好的归并算法相比,它要使用与 n^2 成比例的时间。最后(见参考书目),有人已发明了在与 n 成比例的时间内归并两个顺序表同时只使用较少的、固定数量的额外空间的算法,然而这些算法却非常复杂。

练习 8.7

E1. 一本专业期刊中的一篇文章声明:"这个递归过程(归并排序)花费时间为 $O(n \log n)$,因此在对 256 个数排序时,比前一种方法(插入排序)快 64 倍。"评价这种说法。

E2. 归并中的键比较次数一般较高,因为它不考虑一个表在另一个表之前结束的事实。例

如,第一个表中的所有元素都在第二个表中任何元素之前的情况是可能发生的,因此比较次数仅是第一个表的长度。在本练习中,我们假设两个表中所有的数各不相同,并且这些数的各种排列出现的机率相同。

(1)证明我们的算法在归并长度为2的两个有序表时执行的平均比较次数是8/3。(提示:从有序表1,2,3,4开始,写下将这些数分成长度为2的两个有序表的6种方式,并证明其中4种方式要使用3次比较,另外两种方式要使用2次比较。)

(2)证明在归并长度为3的两个有序表时进行比较的平均次数是4.5。

(3)证明在归并长度为4的两个有序表时进行比较的平均次数是6.4。

(4)使用前面的结果得到归并排序的改进的键比较的总次数。

(5) 证明当 m 趋于无穷时,归并长度为 m 的两个有序表时进行比较的平均次数接近 $2m - 2$。

* * * 固定空间线性时间的归并

E3. (非常具有挑战性)通过在不同数组中创建归并表去归并两个顺序表的简单方法,需要使用与两个表中的元素数目成比例的额外空间,但能够被编写得高效运行,所需的时间与元素的数目成比例。尝试为顺序表设计一种归并方法,它要求尽可能少的额外空间,但仍然能以与表中元素数目成(线性)比例的时间运行。(存在一种解决方案,它仅使用很少的常量的额外空间,见参考书目。)

程序设计项目 8.7

P1. 在你的计算机上实现链表的归并排序,与链式插入排序的实现和测试使用同样的约定和同样的测试数据。对较短和较长的表,也对近乎有正确次序和随机次序的表,比较归并排序和插入排序的性能。

P2. 我们的链表归并排序程序要花费相当多的时间查找每个子表的中央元素,以使得能将表分成两半。实现下面的修改方案,它节省了其中一些时间。重新编写 divide_from 函数,使用一个附加的参数给出原始结点表的长度,用这种方式简化和加速表的再分。在函数 merge_sort()和 recursive_merge_sort()中需要什么修改?

* * * 自然归并排序

P3. 我们的归并排序几乎不关心原始表是否已部分拥有正确的次序,在自然的归并排序(natural mergesort)中,这个表在键的递增序列的尾部被分成了表,而不是任意地在表的半程点上。这个练习要求实现两种版本的自然归并排序。

* * * 一个有序表

(1)在第一个版本中,仅对原始表遍历一次,并且仅使用两个子表。只要键的次序是正确的,这些结点就被放在第一个子表中,当发现一个失序的键后,终止第一个子表并开始第二个子表。当再次发现另一个失序的键后,终止第二个子表,并将第二个子表归并到第一个子表中,然后再次重复第二个子表的构造并将它归并到第一个表中。当到达原始表的结尾时,排序结束。第一个版本易于设计,但随着它的进行,第一个子表可能比第二个子表长很多,导致此函数的性能退化到插入排序的水平。

* * * 几个有序表

(2)第二个版本保证被归并的子表的长度接近相等,因而完全发挥分而治之的优势。这

个方法维持一个(小的)辅助数组,此数组包含①尚未排序的有序子表的长度和②指向尚未排序的有序子表的头结点的指针,这个数组中的元素应根据子表的长度保持有序。当从原始表中分出每个(自然有序的)子表时,这些子表被放入辅助数组中,如果在这个数组中有另一个子表,它的长度在新的子表的一半和两倍之间,则将这两个表归并,并重复此过程。当原始表被穷尽时,归并这个数组中任何剩余的子表(较小的长度优先),排序结束。

不必恐惧归并子表的准则中 2 的比率,它的选择仅保证辅助数组中的元素数目不能超过 $\lg n$(证明它!)。较小的比率(要求大于 1)会使辅助表格更大,而较大的比率会减少分而治之的优势。用测试数据试验以找出可用的较好的比率。

*** 顺序归并排序

P4. 为顺序表设计一个版本的归并排序,困难在于设计一个函数去归并顺序存储的两个有序表。除两个表所需的空间外,有必要使用一些附加的空间。最简单的解决方案是使用两个数组,每个都大得足够容纳两个原始表中的全部元素。这两个有序子表占据同一数组中的不同部分。当对它们进行归并时,在第二个数组中建立新的表。在归并完成后,如果需要的话,新的表能被复制回到第一个数组中。换句话说,在下一阶段能够颠倒这两个数组的作用。

8.8 顺序表的快速排序

我们现在介绍快速排序方法,此方法首先将顺序表划分成较低的和较高的两个子表,较低子表中的所有键都比所选定的某个支点(pivot)的键小,而较高子表中的所有键都比该支点的键大。快速排序也可以在链表上很简单地实现,这部分工作将作为一个设计项目来完成。但快速排序最重要的应用是对顺序表,它对顺序表的排序速度很快。与顺序归并排序相比,快速排序不需要另外使用一个辅助数组而耗费相当多的额外空间,也不需要为了减少额外空间而投入大量的编程工作去实现一个复杂而困难的归并算法。

8.8.1 主函数

设计顺序快速排序的任务是写一个算法,它使用一个支点来划分顺序表中的元素,并在顺序表中交换这些元素,使得键比支点键小的那些元素都排在支点前面,支点后面是与该支点具有相等键的元素,再后是那些具有较大键的元素。我们使用变量 pivot_position 来存放支点在划分表中的位置。

由于划分后的子表仍然以适当的相对位置保存在同一数组中,完全不必再合并所保存的子表,因此在函数中省略了这最后一步。

为了将排序函数递归地应用于子表,需要将子表的边界 low 和 high 作为此函数的参数。然而,我们以前的排序函数都是不带参数的,因此为保持叙述的一致性,我们在函数 recursive_quick_sort 中实现递归,而采用方法 quick_sort 来调用 recursive_quick_sort,quick_sort 不带参数。

*** 主函数 quick_sort

```
template <class Record>
void Sortable_list<Record> :: quick_sort()
/* Post: The entries of the Sortable_list have been rearranged so that their keys are
        sorted into nondecreasing order.
  Uses: The Contiguous List implementation of Chapter 6, recursive_quick_sort. */
{
  recursive_quick_sort(0, count-1);
}
```

真正的顺序表快速排序函数如下：

* * * 递归函数 recursive_quick_sort

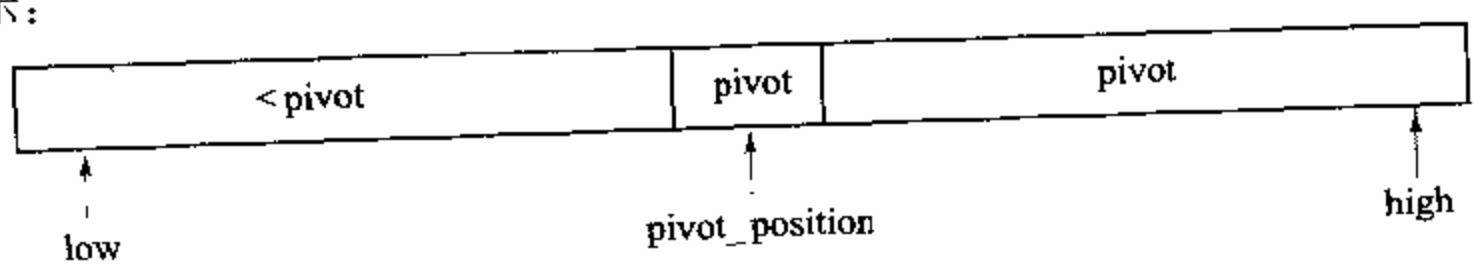
```
template <class Record>
void Sortable_list<Record> :: recursive_quick_sort(int low, int high)
/* Pre:  low and high are valid positions in the Sortable_list.
  Post: The entries of the Sortable_list have been rearranged so that their keys are
        sorted into nondecreasing order.
  Uses: The Contiguous List implementation of Chapter 6, recursive_quick_sort, and
        partition. */
{
  int pivot_position;
  if(low<high){   //  Otherwise, no sorting is needed.
    pivot_position=partition(low, high);
    recursive_quick_sort(low, pivot_position-1);
    recursive_quick_sort(pivot position+1, high);
  }
}
```

8.8.2 表的划分

我们现在来构造 partition 函数。可以使用几种策略实现(其中之一将作为练习)，有的策略比我们所设计的算法要快，但却很复杂并且容易出错。我们所设计的算法更加简单而且易于理解，当然也不慢。实际上，在所有的算法中，它所进行的键比较的次数可能是最少的。

1. 算法设计

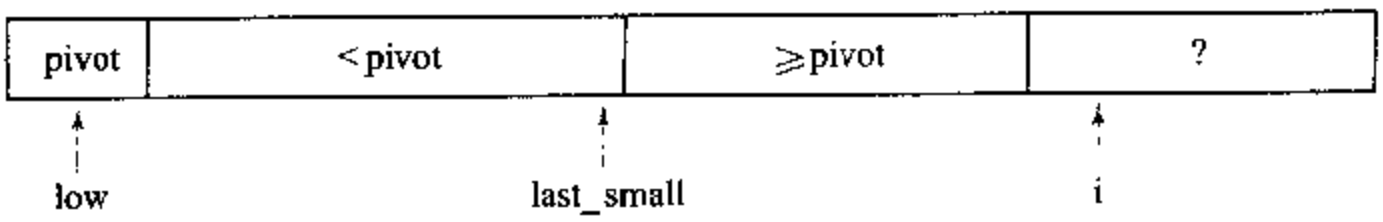
给定一个支点 pivot，我们必须重新排列表中的元素并计算一个下标 pivot_position，使得支点位于 pivot_position，在此下标左边的所有元素的键均小于 pivot，在其右边的所有元素均有较大的键。为了允许存在多于一个元素具有等于 pivot 的键，我们要求 pivot_position 左边的元素的键均严格小于 pivot，而其右边的元素的键可大于或等于 pivot，如下图所示：



* * * 目标(后置条件)

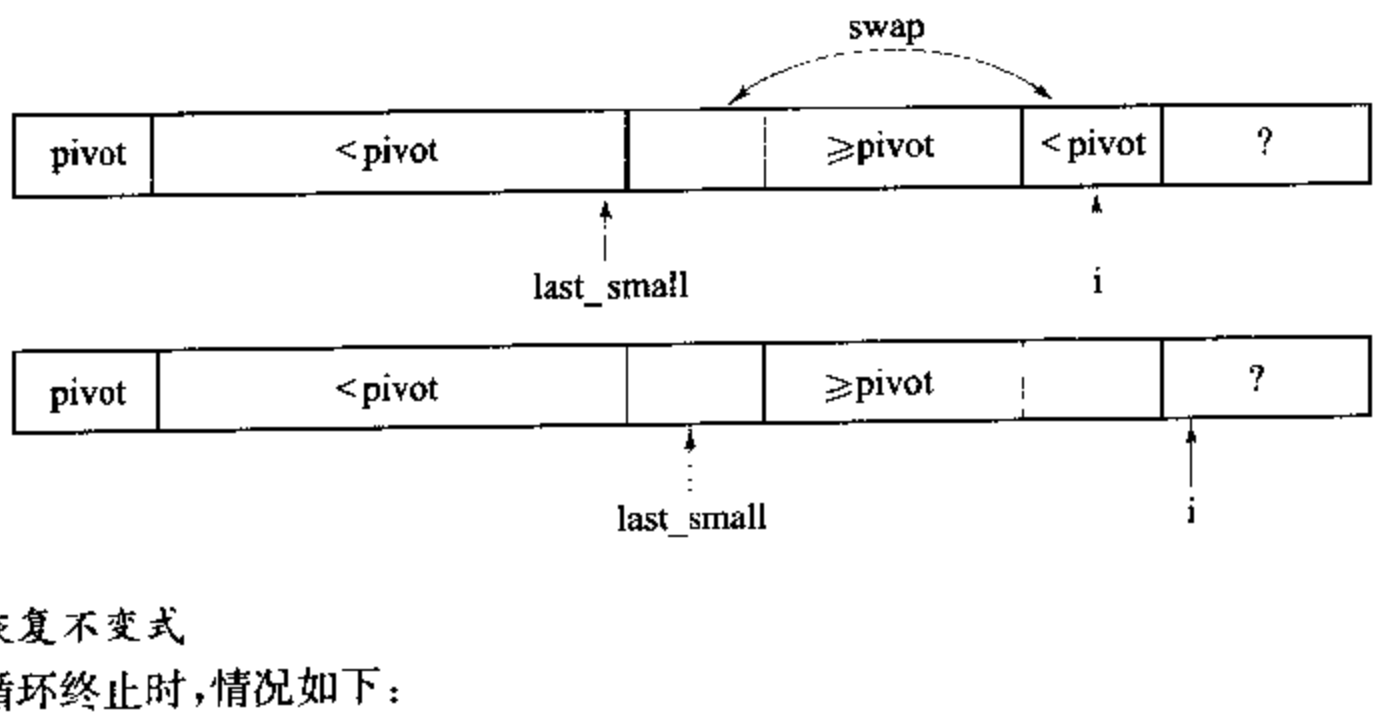
为了按照上述方式记录这些元素，必须将每个键与 pivot 进行比较，我们使用一个 for

循环(在变量 i 上运行)来完成。我们使用第二个变量 last_small,使得在 last_small 位置上或其之前的所有元素有小于 pivot 的键。假设 pivot 从第 1 个位置开始,我们将其暂存于此处,则在循环当中,顺序表具有下列属性:



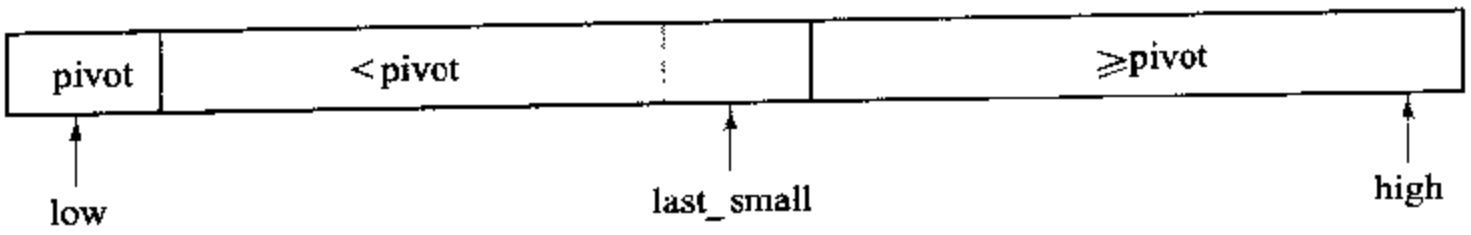
* * * 循环不变式

当函数检查到第 i 个元素时,可能存在两种情况:如果元素大于或等于 pivot,则递增 i,且顺序表保持所要求的属性;如果元素小于 pivot,则我们通过递增 last_small 并将那个元素(它们中的第一个至少是 pivot)与第 i 个元素交换来恢复此属性,如下图所示:



* * * 恢复不变式

当循环终止时,情况如下:



* * * 最终位置

我们然后只需将 pivot 从 low 位置交换到 last_small 位置,即可得到期望的最终排序序列。

2. 支点的选择

* * * 在中央位置选择支点

并非一定要选顺序表的首个元素作为支点,我们可以选择表中任意一个元素并在划分表的循环开始前将它与表中第一个元素交换。实际上,首个元素作为支点经常是一种拙劣的选择,因为一旦此顺序表是已经排过序的,则不会有其他元素再小于此支点,这将导致其中一个子表为空表。因此,通常是在表的近中央位置处选择一个支点,希望这种选择可以把关键字划分在支点两边约各占一半。

3. 编码

按照以上这些抉择，我们得到下列函数，其中用到了 8.3.2 节中的交换函数。为引用方便，我们也在函数中引入循环迭代期间所拥有的属性作为一个断言(循环不变量)。

```
template <class Record>
int Sortable_list<Record> ::partition(int low, int high)
/*Pre: low and high are valid positions of the Sortable_list, with low<=high.
  Post:The center(or left center) entry in the range between indices low and high of the
       Sortable_list has been chosen as a pivot. All entries of the Sortable_list
       between indices low and high, inclusive, have been rearranged so that those with
       keys less than the pivot come before the pivot and the remaining entries come
       after the pivot. The final position of the pivot is returned.
  Uses:swap(int i, int j) (interchanges entries in positions i and j of a Sortable_
       list), the contiguous List implementation of Chapter 6, and methods for the
       class Record.*/
{
  Record pivot;
  int i,                  //  used to scan through the list
    last_small;           //  position of the last key less than pivot
  swap(low,(low+ high)/2);
  pivot=entry[low];       //  First entry is now pivot.
  last_small=low;
  for (i=low+ 1;i<=high;i++)
/*At the beginning of each iteration of this loop, we have the following conditions:
        If low<j<=last_small then entry[j].key<pivot.
        If last_small<j<i then entry[j].key>=pivot.*/
  if (entry[i]<pivot){
    last_small=last_small+1;
    swap(last_small,i);       //  Move large entry to right and small to left.
    }
  swap(low,last_small);       //  Put the pivot into its proper position.
  return last_small;
}
```

8.8.3 快速排序算法的分析

现在来详细分析快速排序算法，以判断它何时效果很好，何时效果又较差，以及它所执行的计算量。

1. 支点的选择

* * * 最坏情况

我们将表中间位置上的键作为支点的选择是随意的，这一选择可能会成功地将表分成近似的两半，但也可能令人遗憾，使得其中一个子表比另一个子表长很多。在练习中将会考虑选择支点的一些其他方法。对如下的顺序表，我们的方法会产生一种极端情况，其中被选中的每个支点都将是子表中的最大键：2 4 6 7 3 1 5。

我们用文中的 partition 函数来检验它。当快速排序应用于此表时，算法显得很不恰当，

因为在首次递归时非空子表的长度为6,第2次递归时为5,以此类推。

如果我们将每个子表中的第一个键或最后一个键选作支点,则当这些键已按正序或反序排列时也会产生极端情况,而这些顺序比某个随机顺序更可能发生,因此选择第一个或最后一个键作为支点可能会带来问题。

2. 比较和交换次数

* * * 总比较次数

现在来判断顺序快速排序所进行的比较和交换的次数。设$C(n)$是对长度n的顺序表应用快速排序所进行的键比较次数,$S(n)$是元素交换次数,则有$C(1)=C(0)=0$。划分函数将支点同表中的每个其他元素正好比较1次,因此函数partition恰好需要进行$n-1$次键比较。如果它所生成的其中一个子表的长度为r,则另一个子表的长度为$n-r-1$。在两个递归调用中所进行的比较次数将分别为$C(r)$和$C(n-r-1)$,因此我们有:

$$C(n) = n-1+C(r)+C(n-r-1)$$

为求解此方程,我们需要知道r。实际上,我们的符号有点欺骗性,因为C()的值不仅依赖于表的长度,同时也依赖于表中元素的精确顺序,而按照此顺序,不同的情况会得到不同的答案。

3. 最坏情况下的比较次数

* * * 递推关系

首先,考虑最坏情况下的比较。我们已看到这种情况会在支点完全未能划分顺序表的时候发生,其中的一个子表有$n-1$个元素,而另一个则为空。在此情况下,由于$C(0)=0$,我们得到$C(n)=n-1+C(n-1)$。这种形式的表达式被称为一个递推关系,因为它的结果由其前一种情况的结果得到。求解此递推式意味着为$C(n)$找到一个另一边不含C()的方程。求解这个递推关系需要用到不同的方法(有时会比较困难),但在本例中可以简单地通过自底向上而非自顶向下的方法来求解它:

$$\begin{aligned}
&C(1)=0.\\
&C(2)=1+C(1) && =1\\
&C(3)=1+C(2) && =2+1\\
&C(4)=1+C(3) && =3+2+1\\
&\vdots && \vdots\\
&C(n)=n-1+C(n-1) && =(n-1)+(n-2)+\cdots+2+1\\
& && =n^2/2-n/2
\end{aligned}$$

在此计算中,我们利用定理A.1来获得$1\sim n-1$的整数和。

* * * 选择排序

选择排序大约需要进行$n^2/2-n/2$次键比较,太多的键比较是选择排序的一个弱点(与插入排序相比)。因此在最坏情况下,快速排序与选择排序的最坏情况效果同样较差。

4. 最坏情况下的交换次数

＊＊＊答案

我们接下来判断快速排序在最坏情况下交换元素的次数。划分函数在其循环内部要对每个比支点小的元素进行 1 次交换，在循环外部进行 2 次交换。在最坏情况下，支点为表中的最大键，因此划分函数会进行 $n+1$ 次交换。设 $S(n)$ 表示在长度为 n 的表上进行交换的总次数，则在最坏情况下我们可以得到递推式 $S(n)=n+1+S(n-1)$。划分函数仅在 $n\geqslant 2$ 时被调用，在最坏情况下 $S(2)=3$。因此与计算比较次数一样，我们通过向下计算来求解此递推式，得到最坏情况下的交换次数：

$$S(n)=(n+1)+n+\cdots+3=(n+1)(n+2)/2-3=0.5n^2+1.5n-1$$

5. 与插入排序和选择排序的比较

＊＊＊低劣的最坏情况性能

在最坏情况下，顺序插入排序必须进行其平均情况下约 2 倍的元素比较和赋值，对每次操作来说共有 $0.5\,n^2+O(n)$。快速排序中的每次交换要求 3 次元素赋值，因此快速排序在其最坏情况下要进行 $1.5\,n^2+O(n)$ 次赋值，换言之，对较大的 n，约为选择排序的 3 倍。但同选择排序相比，移动元素是插入排序的一个弱点。因此在其最坏情况下，快速排序（所谓的）比插入排序的不足之处更差，并且对于键比较，它与插入排序的不足之处一样差。的确，在最坏情况分析中，快速排序是一个灾难，它的名字不亚于虚假广告。

＊＊＊极好的平均情况性能

然而必然存在某些其他原因，使得快速排序没有在不久前被扔到无用程序的垃圾堆中。此原因就在于快速排序被应用到随机顺序表时的平均性能是任何已知的排序方法中（使用键比较并应用于顺序表）最好的算法之一。

8.8.4　快速排序的平均情况分析

为进行平均情况分析，我们假设表的所有可能的次序出现的机率相同，而且为简单起见，仅取整数 1～n 作为键。

1. 交换次数

当我们在函数 partition 中选择支点时，表中的任何一个键都有同等的可能。设 p 表示被选作支点的任意键，则在划分后，由于键 $1,\ldots,p-1$ 均在 p 的左边，而 $p+1,\ldots,n$ 均在 p 的右边，从而保证键 p 是在位置 p 上。

在对 partition 的一次调用中所进行的交换次数为 $p+1$，包括在循环中为小于 p 的 $p-1$ 个键各进行一次交换，以及在循环外部的两次交换。我们用 $S(n)$ 表示快速排序在长度为 n 的表上所进行的平均交换次数，并用 $S(n,p)$ 表示长度为 n 的表上当第一次划分的支点是 p 时的平均交换次数。我们刚才已指出，对 $n\geqslant 2$，$S(n,p)=(p+1)+S(p-1)+S(n-p)$。由于 p 是随机的，我们现在必须取这些表达式的平均，从 $p=1$ 到 $p=n$ 将它们相加并除以 n。此计算用到了整数和的公式（定理 A.1），结果为：

$$S(n)=\frac{n}{2}+\frac{3}{2}+\frac{2}{n}(S(0)+S(1)+\cdots+S(n-1))$$

2. 求解递推关系

求解此递推关系的第一步是要注意到，如果我们是在对长度为 $n-1$ 的表排序，假设 $n \geqslant 2$，则我们可以用 $n-1$ 替换 n 后得到同样的表达式：

$$S(n-1)=\frac{n-1}{2}+\frac{3}{2}+\frac{2}{n-1}(S(0)+S(1)+\cdots+S(n-2))$$

将第一个表达式乘以 n，第二个表达式乘以 $(n-1)$，然后相减，得到：

$$nS(n)-(n-1)\ S(n-1)=n+1+2S(n-1)$$

或：

$$\frac{S(n)}{n+1}=\frac{S(n-1)}{n}+\frac{1}{n}$$

我们现在可以像求解以前的递推关系一样从底部开始求解这个表达式，结果是：

$$\frac{S(n)}{n+1}=\frac{S(2)}{3}+\frac{1}{3}+\cdots+\frac{1}{n}$$

在 A.2.8 节研究了整数倒数的和，在那里证明了：

$$1+\frac{1}{2}+\cdots+\frac{1}{n}=\ln n+O(1)$$

这个和与我们想要的和之间的区别仅是常数的约束，因此我们得到：$S(n)/(n+1)=\ln n+O(1)$，或最后有：$S(n)=n\ln n+O(n)$。

为将这个结果与其他排序方法的结果进行比较，我们指出 $\ln n=(\ln 2)(\lg n)$，并且 $\ln 2\approx 0.69$，因此 $S(n)=0.69(n\lg n)+O(n)$。

3. 比较次数

在长度为 n 的表上对 partition 函数的一次调用恰好进行 $n-1$ 次比较，平均情况下所进行的比较次数的递推关系与交换次数的递推关系之间只有一点区别：在 partition 函数中有 $n-1$ 次比较，而不是 $p+1$ 次交换。因此长度为 n 的表上支点为 p 的比较次数 $C(n,p)$ 的第一个递推式是 $C(n,p)=n-1+C(p-1)+C(n-p)$。当对这个表达式从 $p=1$ 到 $p=n$ 取平均时，我们得到：

$$C(n)=n+\frac{2}{n}(C(0)+C(1)+\cdots+C(n-1))$$

由于键比较次数 $C(n)$ 的这个递推式与 $S(n)$ 的递推式的不同之处只是后者中有 $\frac{1}{2}$ 的因子，应用求解 $S(n)$ 的同样的步骤将得到 $C(n)\approx 2n\ln n+O(n)\approx 1.39n\lg n+O(n)$。

8.8.5 与归并排序的比较

＊＊＊键比较

刚刚完成的计算表明，平均来说，快速排序的键比较比下限要求的多 39%，因此也就是比归并排序多 39%。当然原因在于归并排序被精心设计成将表分成长度基本相等的两半，而快速排序的子表的长度事先无法预知，这样，快速排序的性能可能严重退化，但在

实际中这种情况未必出现，因此，对差的性能和好的性能的比较次数平均后产生刚才得到的结果。

* * * 数据移动

* * * 优化

关于数据移动，我们不对归并排序导出详细的信息，因为我们主要对链式版本感兴趣。但是，如果我们考虑顺序归并排序的版本，它在附加数组中建立归并子表，并且每一趟反转数组的使用，那么很明显，在递归树的每一层上，所有 n 个元素要从一个数组复制到另一个数组中。递归树的层数为 $\lg n$，因此接下来得到顺序归并排序中的元素赋值次数为 $n\lg n$。另一方面，对快速排序来说，我们得到平均情况下大约 $0.69n\lg n$ 的交换次数。好的(机器语言)实现应在两次赋值中完成一次元素交换，因此再次得出：快速排序的元素赋值比归并排序多 39%。但在练习中叙述了另一种划分函数，它平均只有我们设计的版本的 1/3 的交换。因此采用这种细化办法后，顺序快速排序的数据元素赋值可能比顺序归并排序的一半还少。

练习 8.8

E1. 如果表中所有的键都相等，则快速排序函数(如课本中提供的)将如何运行?

E2. 源自 Knuth)描述一种排列顺序表的算法，它的键是实数，使具有负数键的所有元素都排在前面，后面跟着非负的键，最后的表不需完全有序。使你的算法进行尽可能少的元素移动和比较，不要使用辅助数组。

E3. (源自 Hoare)假设我们并不想排序，只是想从给定的长度为 n 的表中找出第 m 个最小的键。说明如何将快速排序用于这个问题，不用做完全的排序。

E4. 给定一个整数表，设计一个函数，它与划分函数类似，它将重排整数以使所有偶数号位置上的所有整数是偶数，或者奇数号位置上的所有整数为奇数。(你的函数应提供证明，虽然可能不会同时达到这两个目标，但二者之一总能达到。)

E5. 在快速排序中选择支点的一种不同方法是取表的第一个键、最后一个键和中央的键三者的中间值。为实现这种选择，描述需要对函数 quick_sort 做出的修改。需要做多少额外的计算？对 $n=7$，查看下面的键序 $1,2,\ldots,7$ 会使算法进入最坏情况，这种情况比原始算法的最坏情况好多少?

* * * 均值排序

E6. 选择支点的一种不同方法是将表中所有键的均值(平均)取作支点，结果得到的算法称为均值排序(meansort)。

(1)编写函数实现均值排序。必须修改 partition 函数，因为键的这个均值不必是表中的某个键。在第一趟，可任意选择支点，当在随后划分键时，保持两个子表的和及个数，因此不用对表进行额外的扫描即可计算出子表的均值(这个均值将是新的支点)。

(2)在均值排序中，键的相对大小决定划分后子表近似相等的程度，除了要对交换次数计数外，键的初始次序并不重要。根据两个子表的相对大小，均值排序的最坏情况可能有多坏？找出产生均值排序的这种最坏情况的 n 个整数的集合。

E7. (要求初等概率的理论)选择支点的一种好办法是使用伪随机数生成器去为 recursive_

quick_sort 的每次调用的下个支点选择位置。利用这些选择是独立的这一事实，找出快速排序发生最坏情况的概率。(1)用 $n=7$ 完成这个问题；(2)用一般的 n 完成这个问题。

* * * 优化 partition

E8. 以更多的一些键比较为代价，可重新编写 partition 函数，使它的交换次数减少到 1/3，平均从 $\frac{1}{2}n$ 减少到 $\frac{1}{6}n$。其思想是用两个下标从表的两端向中央移动，并仅在低位置上发现较大的键和在高位置上发现较小的键时执行交换。这个练习略述了这个函数的设计。

(1)建立两个下标 i 和 j，并维持一种不变性，即位置 i 前的所有键都小于或等于支点，而位置 j 之后的所有键都大于支点。为简单起见，将支点交换到第一个位置上，并从第二个元素开始划分。编写一个循环，只要不变性成立，就增加 i，另一个循环只要不变性成立就减少 j。你的循环也必须保证下标不超出范围，可以检查 $i \leqslant j$。当找到一对元素，各在错误的一边，则将它们交换并重复循环。这个外部循环的终止条件是什么？最后支点能被交换到其合适的位置上。

(2)使用不变性，验证你的函数运行正确。

(3)证明循环中执行的每个循环将两个元素放到其最终位置上，并从这一点说明，对长度为 n 的表，函数在最坏情况下最多进行 $\frac{1}{2}n+O(1)$ 次交换。

(4)在划分后，如果支点位于位置 p，则此函数所做的交换次数近似于原来在支点处或支点前的 p 个位置中的某一个上的元素数，但它们的键大于或等于这个支点。如果这些键是随机分布的，那么特定键大于或等于支点的概率是 $\frac{1}{n}(n-p-1)$。证明这种键的平均数目，并因此证明交换的平均次数近似于 $\frac{p}{n}(n-p)$。通过从 $p=1$ 到 $p=n$ 取这些数的平均，证明交换的次数近似于 $\frac{n}{6}+()(1)$。

(5)可以将支点键用作停止循环的监视哨，从而消除检查确认 partition 中的下标 i 和 j 停留在界限之内的需要。在你的函数中实现这种方法，确信去验证你的函数在所有情况下运行正确。

(6)(源自 Wirth)考虑下面这个简单而“明显”的方法，将支点作为监视哨，编写一个循环：

```
do{
  do{i=i+1;}while(entry[i]<pivot);
  do{j=j-1;}while(entry[j]>pivot);
  swap(i,j);
}while(j>i);
```

找出一个键表，使这个版本对它是失败的。

程序设计项目 8.8

P1. 在你的计算机上，实现快速(顺序表)排序，并用 8.2 节项目 P1 的程序测试它。比较它

和以前学习的所有其他排序方法的性能。

P2. 为链表编写一个快速排序版本，将它集成到 8.2 节项目 P1 的链式版本的测试程序中，并比较它与使用链表的其他排序方法的性能。

* * * 链式快速排序

使用子表的第一个元素作为划分的支点，链表的 partition 函数比顺序表稍微简单些，因为不需要关心数据的移动。为划分一个子表，仅需要遍历它，在遍历过程中删除每个元素，然后依据这个元素的键相对于支点的大小而将它增加到其中一个表中。

然而，由于 partition 现在产生两个新的表，因此需要一个附加的短函数将有序的子表重新合并为单个的链表。

P3. 由于涉及较多的系统开销，对短表来说，快速排序可能比简单方法要差些。通过实验找出一个值，使得对随机次序的表，平均起来，快速排序比插入排序更有效。编写一个混合的排序函数，它从快速排序开始，当子表足够短时，再转换到插入排序。确定哪一种方法更好：(1)在递归函数内部切换；或者(2)当子表足够短时，终止递归调用去改变方法，然后在此过程的结尾处，对整个表运行一次插入排序。

8.9 堆和堆排序

虽然快速排序的一般性能很出色，但它也存在缺点，某些种类的输入会导致它性能恶劣。我们将在本节学习的另一种排序方法克服了这个问题，这种算法称为堆排序(heapsort)，即使在最坏情况下，它对长度为 n 的顺序表的排序仍具有 $O(n \log n)$ 次元素的比较和移动。因此它达到的最坏情况界限比快速排序的要好，而且对顺序表来说，它比归并排序好，因为除了待排序的表以外，它只需要很少的常量级的空间。

* * * 社会阶层

堆排序基于反映了社会阶层中的强弱次序的树结构。将公司管理的组织结构想像成一棵树，树的顶部是总裁。当总裁退休时，副总裁竞争最高的职位，其中有一位会获得提升并造成一个职位空缺。只要出现了这个职位空缺，一般高级主管就会竞争这个提升机会。现在(非常不现实地)想像这个公司总是通过给最有价值的雇员即总裁发放养老金使其退休，因此连续在顶部出现职位空缺，雇员竞争这次提升，并且当每个雇员到达这个“堆顶”时，他的职位再次成为空缺。有了这种想像，我们得到了这种排序方法的本质思想。

8.9.1 二路树作为表

我们从一棵完全 2-树开始，如图 8.15 中显示的这棵树，并且对顶点编号，从树根开始标记为 0，在每层都从左到右编号。

现在通过存储由标记显示的位置中的每个结点，将这棵 2-树放入一个表中。我们得出结论：

* * * 查找孩子

如果树的根在表的位置 0，则在表中位置 k 的结点的左、右孩子分别在位置 $2k+1$ 和 $2k+2$ 中。如果这些位置超过了表的尾部，则这些孩子不存在。

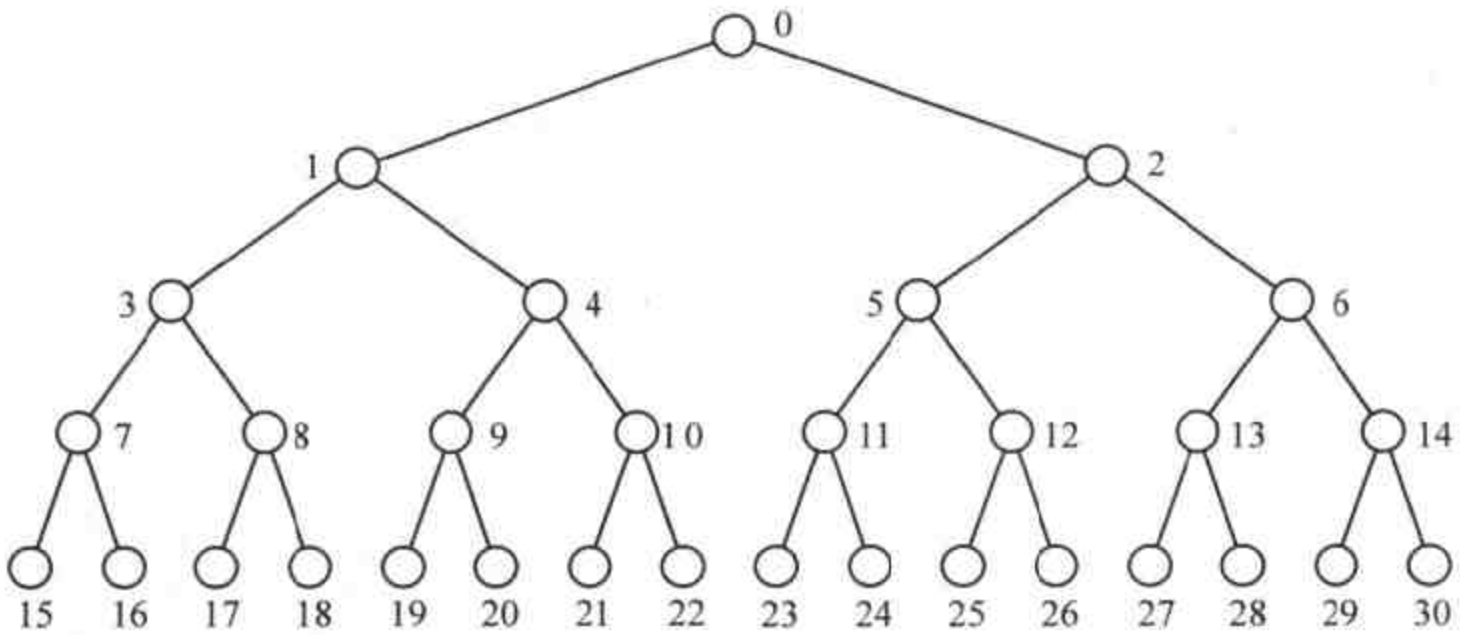


图 8.15 有 31 个顶点的完全 2-树

用这种思想，能够定义我们关于堆的用意。

定义 堆是一种表，它的每个元素包含一个键，并且对表中所有的位置 k，位置 k 上的键至少与位置 $2k$ 和 $2k+1$ 中的键同样大，假设这些位置都在表中存在。

这样，堆就类似于社会阶层，其中每个雇员（除非他们是在堆的底部）监督其他两个雇员。

为了解释堆的用法，我们将画出像 8.16 这样的图去说明层次关系，但运行在堆上的算法总是将它们作为特殊种类的表处理。

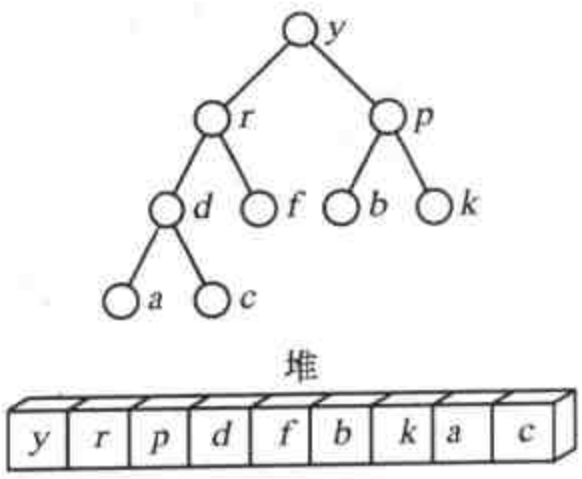


图 8.16 作为树和作为表的堆

注意堆确实不是一种有序表，实际上第一个元素必须拥有堆中的最大的键，而在有序表中第一个键却是最小的。在堆中，如果 $k>1$，位置 k 和 $k+1$ 之间没有必要的次序。

注释 许多C++手册和课本将动态内存使用的内存区称为“堆”，单词“堆”的这种用法与目前的定义没有关系。

8.9.2 堆排序的设计

1. 方法

* * * 两阶段函数

堆排序按两阶段进行：首先，我们必须安排表中的元素，使它们满足堆的需要(类似于组织社会阶层)。其次，我们重复地从堆顶删除并提升另一个元素取代它。

对第二个阶段，我们回想树根(它是表的第一个元素，也是堆顶)有最大的键，这个键位于表的尾端，因此我们将第一个元素移到最后的位置，代替元素 current，然后递减计数器 last_unsorted；last_unsorted 记录表的未排序部分的长度，因此排除了对最大的元素进一步排序的可能性。然而，从最后位置删除的元素 current 可能不属于堆顶，因此必须将 current 插入到合适的位置上，以便在继续用同样的方式循环前恢复堆的属性。

从这里的描述可以看出，堆排序要求随机访问表的所有部分。因此我们必须确定：

堆排序只适用于顺序表。

2. 主函数

现在按我们的标准约定用C++编写堆排序，从而将它具体化。

```
template <class Record>
void Sortable_list<Record> ::heap_sort()
/*Post: The entries of the Sortable_list have been rearranged so that their keys are
        sorted into nondecreasing order.
  Uses: The contiguous List implementation of Chapter 6, build_heap, and insert_heap .
        */
{
  Record current;          //  temporary storage for moving entries
  int last_unsorted;       //  Entries beyond last_unsorted have been sorted.
  build_heap();            //  First phase: Turn the list into a heap.
  for (last_unsorted=count-1; last_unsorted>0; last_unsorted-- ){
      current=entry[last_unsorted];   //  Extract last entry from the list.
      entry[last_unsorted]=entry[0];  //  Move top of heap to the end
      insert_heap(current, 0, last_unsorted-1);  //  Restore the heap
  }
}
```

3. 例子

在介绍 build_heap 和 insert_heap 这两个函数前，我们先来看一下，对图 8.16 中所示的堆进行排序的前几个阶段中发生了什么，这些阶段显示在图 8.17 中。第一步，将最大的键 y 从表的第一个位置移到最后一个元素，第一个图显示了结果得到的树，其中 y 被移走且不作进一步考虑，原来的最后元素 c，被放在一边作为临时变量 current。为找出如何对堆进行重排并插入 c，我们查看根的两个孩子，其中每个孩子都有比其子树中任何其他元素更大的键，因此这两个元素和 c 三者中的最大者属于树根，因此将 r 提升到堆顶，并在已移走树根的子

树中重复这个过程，因此 d 和 f 中的较大者被插入到 r 原来的位置上。在下一步，我们将比较 current=c 和 f 的两个孩子，但 f 的孩子并不存在，因此停止树中元素的提升，并且 current=c 被插入到先前 f 占据的空位上。

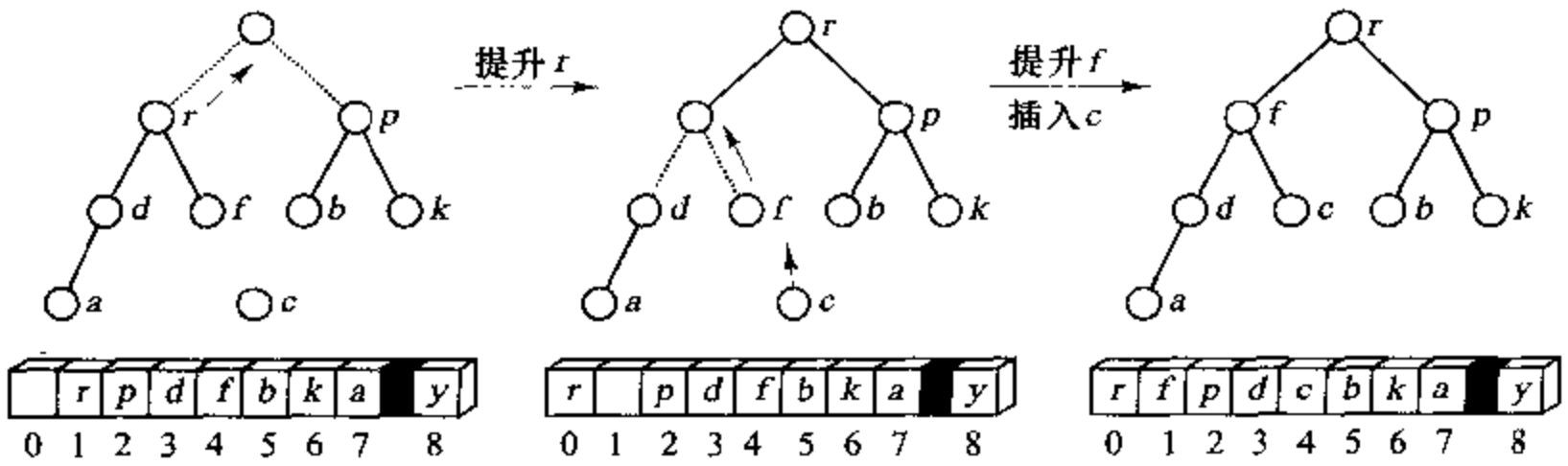


图 8.17 heap_sort 的第一阶段

此时我们即将重复这个算法，再次将堆顶移到表的尾部并恢复堆的属性。图 8.18 中显示了对这个表完全排序的过程中所发生的操作序列。

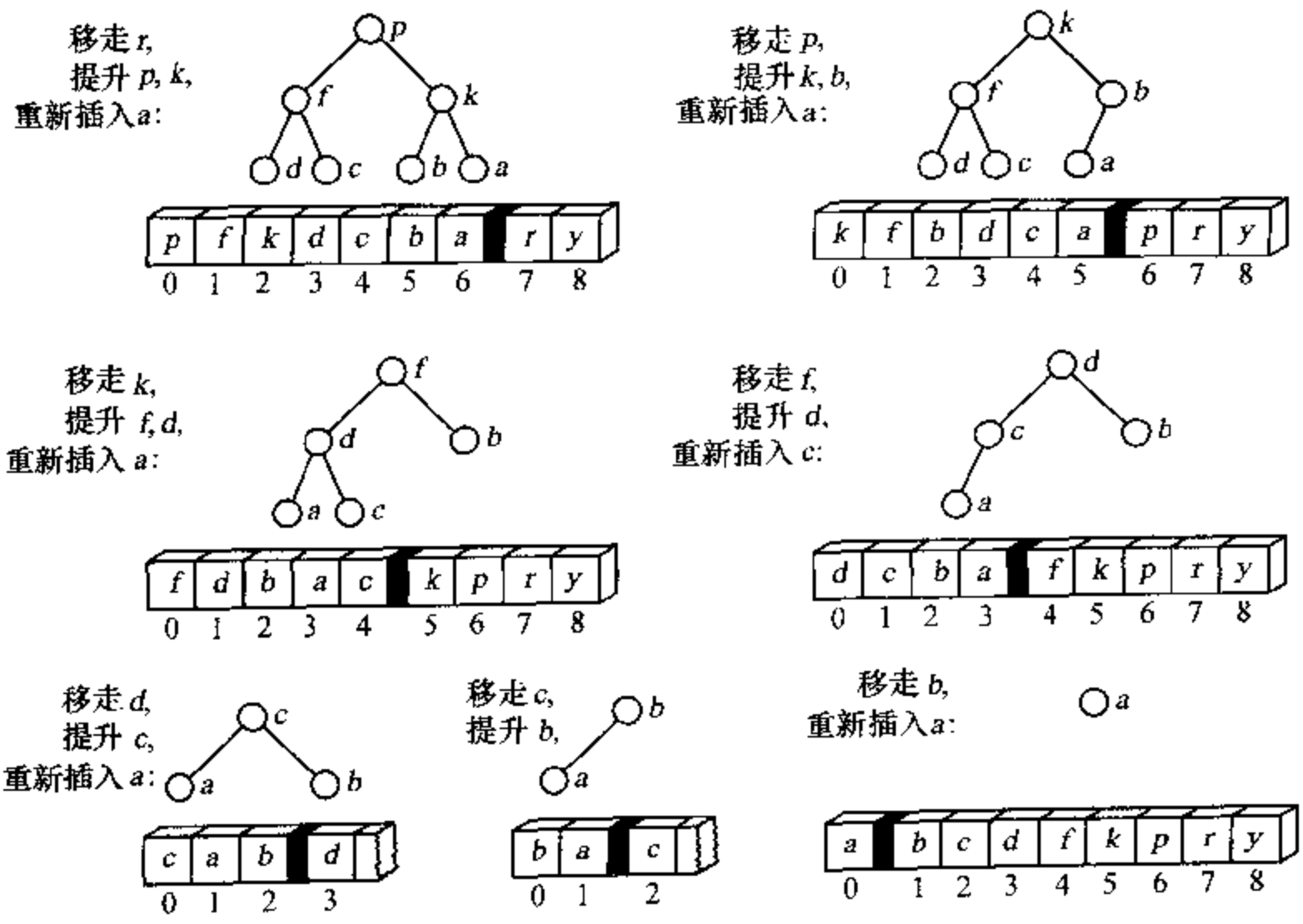


图 8.18 heap_sort 的踪迹

4. 函数 insert_heap

此例近似于向堆中插入元素 current 的正式函数。

```
template <class Record>
void Sortable_list<Record> :: insert_heap(const Record &current, int low, int high)
/*Pre: The entries of the Sortable_list between indices low+1 and high, inclusive, form
```

```
    a heap. The entry in position low will be discarded.
  Post: The entry current has been inserted into the Sortable_list and the entries
        rearranged so that the entries between indices low and high, inclusive, form a
        heap.
  Uses: The class Record, and the contiguous List implementation of Chapter 6.*/
{
  int large;          //  position of child of entry[low] with the larger key
  large=2*low+1;      //  large is now the left child of low.
  while (large <=high){
    if (large<high && entry[large]<entry[large+1])
      large++;        //  large is now the child of low with the largest key.
    if (current>=entry[large])
      break;          //  current belongs in position low.
    else {            //  Promote entry[large] and move down the tree.
      entry[low]=entry[large];
      low=large;
      large=2*low+1;
    }
  }
  entry[low]=current;
}
```

5. 建立初始的堆

* * * 初始化

我们必须说明的剩下的任务是对无序序列的表创建初始的堆。为此，我们首先指出只有一个结点的 2-树自动满足堆的属性，因此不必担心树中的任何树叶，即不必担心表的后一半中的元素。如果从表的中间点开始并从后向前工作，就能够使用函数 insert_heap 去将每个元素插入到包含了所有后面元素的部分堆中，从而创建完整的堆。因此希望的函数只是：

```
template <class Record>
void Sortable_list<Record> ::build_heap()
/*Post: The entries of the Sortable_list have been rearranged so that it becomes a heap.
  Uses: The contiguous List implementation of Chapter 6, and insert_heap.*/
{
  int low;            //  All entries beyond the position low form a heap.
  for (low=count/2-1;low >=0;low--){
      Record current=entry[low];
      insert_heap(current,low,count-1);
  }
}
```

8.9.3　关于堆排序的分析

从这个例子，我们已得出，排序是否有效完全不明显，而实际上，堆排序对短表来说不是一种好的选择。看起来很奇怪，在将较大的键最终放在后端之前，能够通过向表的开头慢慢地移动而进行排序。然而当 n 变大时，这个小的怪癖就变得不重要了。作为少数几种以最小的空间需求而保证在时间 $O(n \log n)$ 内完成顺序表排序的算法之一，堆排序证明了它的价值。

* * * 最坏情况的插入

首先，我们确定 insert_heap 在最坏情况下的工作量。每一趟循环，low+1 的值至少翻倍，因此趟数不会超过 lg ((high+1) / (low+1))，这也是以 entry[low]作为树根的子树的高度。每一趟循环(一般)进行两次比较和一次元素赋值，因此在 insert_heap 中进行比较的次数最多为 2 lg ((high+1) / (low+1))，并且赋值次数为 lg ((high+1) / (low+1))。

* * * 第一阶段

设 $m=\lfloor \frac{1}{2}n \rfloor$(即不超过 $\frac{1}{2}n$ 的最大的整数)，在 build_heap 中，k=low 的值从 $m-1$ 到 0，调用 insert_heap 函数 m 次，因此总的比较次数大约是：

$$2\sum_{k=1}^{m} \lg\left(\frac{n}{k}\right) = 2(m\lg n - \lg m!) \approx 5m \approx 2.5n$$

由 Stirling 近似(附录 A 的定理 A.5)和 $\lg m = \lg n - 1$，我们得到：

$$\lg m! \approx m\lg m - 1.5m \approx m\lg n - 2.5m$$

* * * 第二阶段

* * * 最坏情况下的总次数

类似地，排序和插入阶段，比较次数大约有：

$$2\sum_{k=2}^{n} \lg k = 2\lg n! \approx 2n\lg n - 3n$$

这一项高出初始阶段中的那一项，因此我们得出结论：比较次数为 $2n \lg n + O(n)$。

在 insert_heap 中，每两次比较(大约)要进行一次元素赋值，因此总的赋值次数为 $n \lg n + O(n)$。

总之，我们能够说：

对长度为 n 的表进行排序的最坏情况下，堆排序执行了 $2n \lg n + O(n)$ 次键比较和 $n \lg n + O(n)$ 次元素赋值。

* * * 与快速排序的比较

从 8.8.4 节我们看出，快速排序在平均情况下的相应次数是 $1.39n\lg n + O(n)$ 次比较和 $0.69n \lg n + O(n)$ 次交换，它至少有 $1.39n \lg n + O(n)$ 次元素赋值。因此就比较次数来看，堆排序的最坏情况比快速排序的平均情况稍微差些；就元素赋值来看，堆排序的最坏情况比快速排序的平均情况稍微好些。然而，快速排序的最坏情况是 $\Theta(n^2)$，对较大的 n，它比堆排序的最坏情况差得很远。对堆排序平均情况的分析似乎十分困难，但经验研究表明(与选择排序一样)，其平均情况和最坏情况之间只有相当小的差别，而且堆排序通常花费的时间是快速排序的两倍。

因此，堆排序应被认为是某种保险的方式：平均来看，堆排序的代价比快速排序稍微多些，但堆排序避免了性能灾难性下降，尽管这种情况发生的概率极小。

8.9.4 优先队列

为结束本节，我们简要地提一下堆的另一种应用。

定义 优先队列(priority queue)由元素组成,每个元素包含一个称为此元素的优先级的键。除了通常的创建、清除、取长度、判满、判空操作以外,优先队列只有两种操作:

- 插入一个元素;
- 移走具有最大(或最小)键的元素。

如果元素有相等的键,则任何具有最大键的元素都能被首先移走。

* * * 应用

例如在分时计算机系统中,有大量的任务等待 CPU,其中的一些任务的优先级较高,因此等待 CPU 的任务集合形成一个优先队列。优先队列的其他应用包括与时间有关的事件的模拟(如 3.5 节中的机场模拟)和通过行约简进行的线性方程式的稀疏系统求解方案。

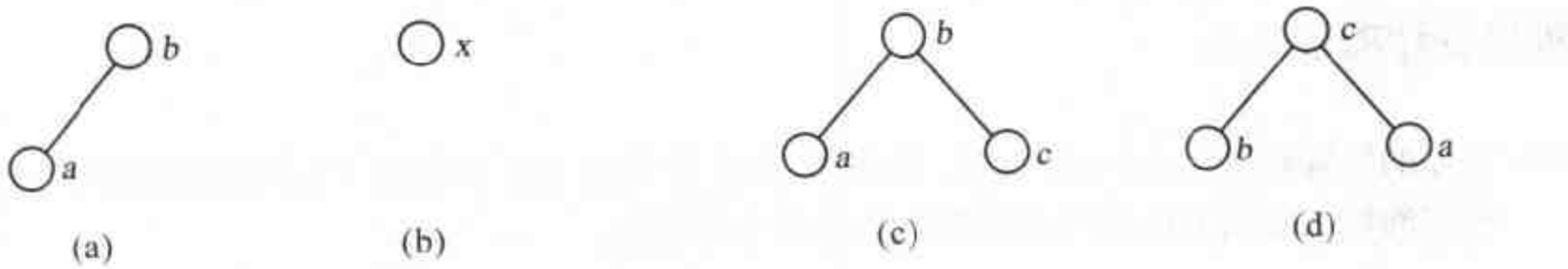
* * * 实现

我们将优先队列表示成一个有序顺序表,在这样的情况下,移走元素是直接的,但插入会消耗与队列元素长度 n 成比例的时间。或者我们能将它表示成一个未排序的表,在这种情况下,插入操作会很快而移走会很慢。

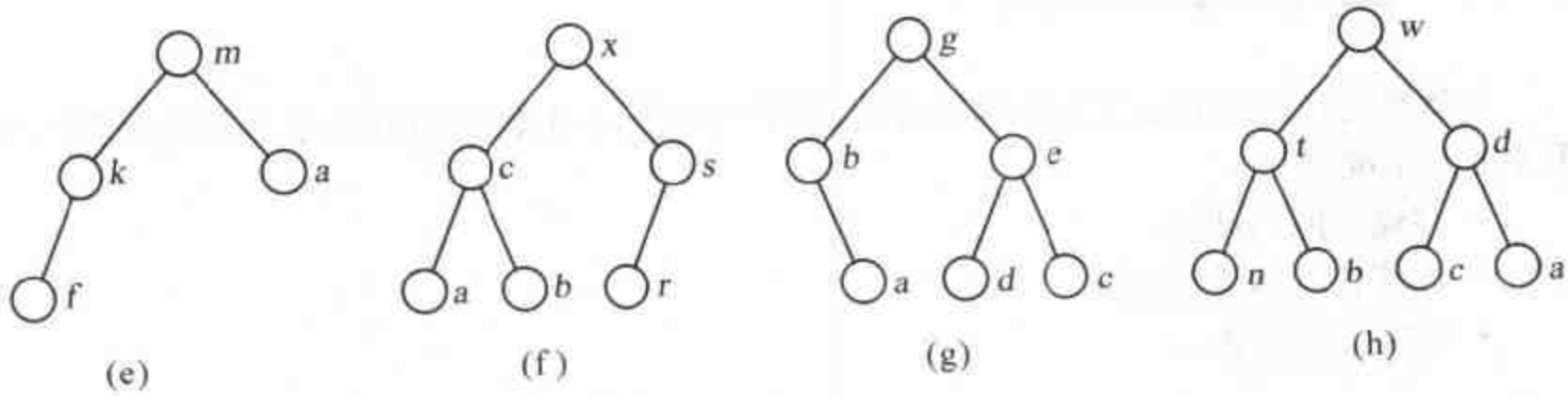
现在考虑堆的属性,具有最大键的元素位于堆顶并能被立即移走。然后需要花费 $O(\log n)$的时间去为剩余的元素恢复堆属性,但如果要立即插入另一个元素,则其中一部分时间可以与插入新元素所需的 $O(\log n)$的时间合并。因此将优先队列表示成一个堆证明对较大的 n 有优势,因为它能有效地在顺序存储中被表达,并且能保证插入和删除只需要对数时间。

练习 8.9

E1. 说明与下列每棵树对应的表符合这种表达,即位置 k 中元素的孩子在位置 $2k+1$ 和$2k+2$ 中。这些树中哪一些是堆?对那些不是堆的,指出表中哪个位置违背了堆属性。



(a) (b) (c) (d)



(e) (f) (g) (h)

E2. 手工跟踪 heap_sort 在下列每个表上的动作,画出表对应的初始树,说明如何将它转换成堆,并说明当从顶部移走每个元素并插入新元素时产生的堆。

(1)下列 3 个单词按字母顺序排序:

triangle square pentagon

(2)对(1)中的3个单词按对应的多边形的边数,以升序排序。

(3)对(1)中的3个单词按对应的多边形的边数,以降序排序。

(4)将下列7个数以升序排序:

26 33 35 29 19 12 22

(5)同样的7个数具有不同的初始顺序,再次以升序排序:

12 19 33 26 29 35 22

(6)将下列14个名字按字母顺序排序:

Tim Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

E3. (1)设计一个函数,将一个新元素插入到堆中而获得一个新的堆。(课文中的insert_heap函数要求树根未被占据,而在本练习中,树根中已包含了具有最大键的元素,它必须保留在堆中,你的函数将增加表中的元素个数。

(2)分析你的函数的时间和空间需求。

E4. (1)设计一个函数,从堆顶删除具有最大键的元素(树根),并恢复因此得到的较小的表的堆属性。

(2)分析你的函数的时间和空间需求。

E5. (1)设计一个函数,从堆中删除具有下标 i 的元素,并恢复所得到的较小表的堆属性。

(2)分析你的函数的时间和空间需求。

E6. 考虑 n 个键的堆,其中对 $0 \leqslant k < n$,x_k 是位置 k 中的键(用顺序表表示)。证明对所有满足 $0 \leqslant k < n$ 的 k,以 k 为根的子树的高度是不超过 $\lg((n+1)/(k+1))$ 的最大整数。(提示:使用对 k 的"后向"归纳法,从树叶开始并返回树根,其中树根为 x_0。)

E7. 定义三元堆(ternary heap)的概念,除了树中非树叶的每个结点有3个孩子以外,它与普通的堆类似。设计一个基于三元堆的排序方法,并分析此排序方法的属性。

程序设计项目 8.9

P1. 在你的计算机上实现(顺序表的)堆排序,将它集成到8.2节项目P1的测试程序中,并将它的性能和以前所有的排序算法的性能进行比较。

8.10 复习:方法比较

本章研究并仔细分析了多种排序方法,或许总结这一工作的最好办法是依次强调3个重要的效率准则:

- 存储空间的使用;
- 计算机时间的使用;
- 程序设计工作。

1. 空间的使用

*** 递归的栈空间

关于空间,我们讨论的大多数算法除原始表所占用的空间外,只使用很少的空间,我们

在表的原始位置上将它重排成有序的表。快速排序和归并排序是例外，在这两种方法中，递归确实需要少量的额外空间去记录尚未排序的子表，但在编写良好的程序中，递归所使用的额外空间为 $O(\log n)$，这与其他目的所需的空间相比是很少的。

最后我们要记住顺序表归并排序的一个主要缺点是简单的版本要求 $\Theta(n)$ 的额外空间，实际上，它等于原始表所占用的空间。

*** 外部排序

*** 外部排序和归并

在许多应用中，待排序的表由于太大而无法全部保存在高速内存中，在这种情况下，必须采用其他的方法。一种经常使用的方法是将表分割成能在高速内存内部进行排序的子表，并在外部归并已经排过序的子表。因此在归并算法方面已做了大量的工作，主要是解决同时归并多个子表的问题。我们不进一步讨论这个主题。

2. 计算机时间

第二个效率准则是计算机时间的使用，我们对自己设计的每个方法都仔细地分析过这个准则。总而言之，插入排序和选择排序的简单方法对长度为 n 的表花费 $\Theta(n^2)$ 的时间，希尔排序更快，其余的方法一般最快，花费 $\Theta(n \log n)$ 的时间。然而快速排序最坏情况的时间是 $\Theta(n^2)$，堆排序从某种形式上说是一种比较保险的方法，它通常比快速排序的代价大，但却避免了性能严重下降。

3. 程序设计工作

第三个准则经常是所有准则中最重要的：这个准则是对程序设计者时间的有效和富有成效的利用。

如果一个表较小，那么设计用来减少计算机时间需求的复杂排序方法通常比简单方法更差或只是在满足其目标上勉强好些。如果一个程序只运行一、两次，并且有足够的计算机时间，则程序设计者花上数天或数周的时间去调查许多复杂的算法，而这些方法最后可能只节省了几秒种的计算机时间，这种做法是很愚蠢的。

*** 老的语言

当采用诸如 Fortran，Cobol 或 Basic 这些不支持递归的语言进行程序设计时，归并排序和快速排序的实现就变得相当地复杂，虽然能像第 5 章认识到的那样，用栈保存变量的值去实现它。

希尔排序在性能上离归并排序和快速排序相差不远，它不需要递归，且易于设计，因此决不要低估希尔算法。

节省程序设计时间是选择简单算法的一个极好的理由，虽然它效率不一定高，但应该始终记住两句告诫：首先，节省程序设计时间决不应当成为编写错误程序的理由，这种程序通常可能运行，但有时会出现异常。墨菲法则不可避免地会成为现实。其次，设计用来仅运行几次并随后就被废弃的简单程序反而经常在应用中找到路子，在开始编写这些程序时，确实并没有想像到有这些应用。在初期不加注意会证明以后的代价将极其昂贵。

对许多应用来说，插入排序证明是最好的选择，它易于编写和维护，并且对短表富有效率。即使对较长的表，如果它们有近乎正确的次序，插入排序也将十分有效。如果这个表是

完全有序的，那么插入排序能尽快地完成这种条件的验证。

4. 统计分析

* * * 中间

算法的最终选择不仅依赖于表的长度、记录的大小和它们在存储中的表示，而且极大程度上依赖于排序前所期望的记录定序方式。从概率和统计的立场去分析算法非常重要。对大多数算法，我们能获得中间(平均)性能的结果，但有关快速排序的经验表明：从一种可能的次序到另一种可能的次序引起的性能变化量也是需要考虑的一个重要因素。

* * * 标准偏差

标准偏差是这种可变性的统计度量。快速排序有极好的平均性能，其标准偏差较小，这意味着性能可能与平均情况差别很小。对选择排序、堆排序和归并排序这样算法，其最佳情况和最坏情况的性能差别较小，这意味着标准偏差也非常小。其他的算法如插入排序，有很大的性能偏差。因此输入表的次序的特定分布是选择排序算法时需要考虑的重要因素，当专业的计算机科学家专注于算法分析时，为了做出智能的决策，他需要知道数理统计学的重要知识。

5. 实验测试

最后，在所有的决定中，我们必须对算法的理论分析辅以实验测试，不同的计算机和编译器会产生不同的结果，因此通过实验了解不同的算法在不同环境中的性能是有益的。

练习 8.10

E1. 将我们所学的排序算法归于下列类别之一：(1)这种方法直到表的一端的元素已排过序才要求访问表另一端的元素；(2)这种方法不要求访问已经排过序的元素；(3)这种方法要求整个过程中都访问表中的所有元素。

E2. 我们所学习的一些排序方法不适合于链表，它们是哪一些？为什么不适合？

E3. 按照用于下标或指针、递归以及复制正被排序的元素所需要的额外存储空间量，对我们所学过的排序方法分级(包括链表和顺序表)

E4. 对下列每种应用，我们所学的哪种方法是较好的选择？为什么？如果表以顺序或链式存储表示，那么对你的选择有什么影响？说明如何影响。

(1)编写一个通用目的的排序程序，它将被许多人用于各种应用中。

(2) 一次性地对 1000 个数排序，在完成这次排序后，不再保存这个程序。

(3)一次性地对 50 个数排序，在完成这次排序后，不再保存这个程序。

(4)在一个较长的程序中间对 5 个元素排序，而且此排序将被这个较长的程序调用几百次。

(5)你有一个 1000 个键的表要在高速内存中排序，而且键比较可以很快地完成，但每次移动一个键时，需要移动磁盘上相应的一个长度为 500 个块(block)的文件，而这种移动是一个缓慢的过程。

(6)有一个长 12 英尺的书架，装满了采用数字分类的计算机科学图书，其中有一些图书

被读者放在了错误的位置上，但这些书几乎都不超过离其正确位置 1 英尺远。

(7)你有一堆 500 张的图书馆索引卡片，它们以随机次序摆放，将它们按字母顺序排序。

(8)一个 5000 个单词的表已按字母顺序排列，但你希望对它进行检查以确信如此，并对所发现的任何失序的单词进行排序。

E5. 讨论将通用排序算法设计成快速排序和希尔排序的混合物的优缺点。从一种排序方法向另一种排序方法转换时将遵循什么准则？哪一种选择对哪种表更好？

E6. 总结本章的排序算法在你的计算机上测试运行的结果，同时包括在练习中编写的任何方法的变种，用图表比较下列诸项：

(1)键比较次数。

(2)元素的赋值次数。

(3)总共的运行时间。

(4)程序的临时存储需求。

(5)程序的长度。

(6)编写和调试此程序所要求的程序设计时间量。

E7. 编写一页长的指南，帮助你的计算机系统用户按其希望的应用选择我们的排序算法中的一种。

E8. 如果两个元素有相等的键，且在排序函数完成时，这两个元素在表中的相对次序保持与排序前一样不变，则这个排序函数称为稳定的。如果一个表已经用一个键排序，现在又要用另一个键排序，并且希望尽可能在新的排序允许的情况下尽量保持原有的次序，那么这时候稳定性就显得很重要。判断本章的排序方法中哪一些是稳定的，哪一些是不稳定的。对那些不稳定的方法，生成一个(尽可能短)的表，表中包含那些具有相等的键而在排序后未保持次序的元素。此外，看看你能否对这些算法进行简单地修改，以使它们变成稳定的。

启示和易犯的错误

1. 许多计算机系统有通用目的的排序例程。如果你能访问这些例程，并且它足够用，那么就使用它们，而不是从头开始编写一个排序程序。
2. 选择排序方法时，考虑键在排序前通常的排列方式、应用的规模、程序设计可用的时间、节省计算机时间和空间的需要、数据结构实现的方式、移动数据的代价和比较键的代价。
3. 分而治之是一种最广泛应用和最强大的设计算法的方法之一。当遇到程序设计问题时，看看能否通过先求解两个(或更多个)形式相同但规模较小的问题而得到问题的解答。如果能，则可以使用分而治之方法明确地表达算法，并用递归进行程序设计。
4. 归并排序、快速排序和堆排序是强大的排序方法，它们比简单方法更难以编程，但它们对大型的表效率更高。仔细考虑应用，确定为实现这些复杂算法之一而需要付出的额外努力是否合理。
5. 优先队列对许多应用来说很重要，而堆提供了优先队列的一种极好的实现。
6. 堆排序是比较保险的方法：它通常比快速排序慢，但它保证排序能够在 $O(n \log n)$ 次键比较内完成，而快速排序一般不行。

复习题

8.2

1. 验证 n 个元素的表是有序的需要多少次键比较?

2. 用 20 个或更少的词解释插入排序如何工作。

8.3

3. 用 20 个或更少的词解释选择排序如何工作。

4. 平均来说,对 20 个元素的表,选择排序比插入排序多进行多少次比较?

5. 选择排序与我们所学的所有其他排序方法相比,其优点是什么?

8.4

6. 希尔排序克服了插入排序的什么缺点?

8.5

7. 如果任何排序方法使用键比较进行决策,则它们对 n 个键排序时必须进行的键比较次数的下界是什么? 同时给出平均和最坏情况的界限。

8. 如果去掉使用比较进行决策的要求,那么下界是什么?

8.6

9. 定义术语"分而治之"。

10. 用 20 个或更少的词解释归并排序如何工作。

11. 用 20 个或更少的词解释快速排序如何工作。

8.7

12. 解释为何归并排序对链表比顺序表更好。

8.8

13. 在快速排序中,我们为什么从表的中央选择支点,而不是从其中一端?

14. 快速排序在平均情况下比最优情况大约多进行多少次键比较? 它在最坏情况下大约进行多少次比较?

8.9

15. 什么是堆?

16. 堆排序如何工作?

17. 比较堆排序最坏情况的性能和快速排序最坏情况的性能,并将它同快速排序平均情况的性能进行比较。

8.10

18. 简单的排序算法在什么时候比复杂的排序算法更好?

进阶参考书目

本章的主要参考书是 D. E. Knuth 的综合系列(第 2 章著书目录的细节)。内部排序的内容在第 3 卷的第 73~180 页。Knuth 比我们更详细地进行了算法分析,他用伪汇编语言编写所有的算法,并详细地计算了操作次数。他研究了我们所学的所有方法,除此之外,还有

更多的一些方法及许多变种。

希尔排序和快速排序的原始的参考书目分别是：D. L. Shell，“A high-speed sorting function，” *Communications of the ACM* 2 (1959)，30-32. 和 C. A. R. Hoare，“Quicksort，” *Computer Journal* 5(1962)，10-15.

归并排序和快速排序的联合的出处是基于下面的工作，它也能用于生成插入排序和选择排序：John Darlington，“A synthesis of several sorting algorithms，” *Acta Informatica* 11 (1978)，1-30.

可以细化归并排序，使它的性能非常接近于最优的下界。这种经过改进的算法的一个例子如下，它的性能离最佳可能只在6%之内：R. Michael Tanner，“Minimean merging and sorting：An algorithm，”*SIAM J. Computing* 7 (1978)，18-38.

以较少的常量的额外空间，在线性时间内运行的一种相对简单的顺序归并算法出现于：Bing-Chao Huang and Michael A. lagston，“Practical in-place merging，” *Communications of the ACM* 31 (1988)，348-352.

在快速排序中划分表的算法是由 Nico Lomuto 发现的，并发表于：Jon L. Bentley，“Programming pearls：How to sort，” *Communications of the ACM* 27 (1984)，287-291.

“Programming pearls”专栏包含许多第一流的算法和程序设计的有益建议，它们被收集在下面这两本书中：Jon L. Bentley，Programming pearls，Addison-Wesley，Reading，Mass.，1986，195 pages. 和 Jon L. Bentley，*More Programming pearls：Confessions of a Coder*，Addison-Wesley，Reading，Mass.，1988，224 pages.

关于快速排序算法的大量分析出于：Robert Sedgewick，“The analysis of quicksort programs，” *Acta Infomatica* 7 (1976-77)，327-355.

关于均值排序(采用键的均值作为支点)的练习来自：：Dalia Motzkin，“MEANSORT，” *Communications of the ACM* 26 (1983)，250-251；27 (1984)，719-722.

堆排序的发现和命名见下面这篇文章：J. W. J. Williams，*Communications of the ACM* 7 (1964)，347-348.

堆和优先队列的算法的一种简单而完整的设计出现于：Jon L. Bentley，“Programming pearls：Thanks，heaps，” *Communications of the ACM* 28 (1985)，245-250.

当然，在计算机的基本应用中有大量概率和统计知识，一本经典的初等概率和统计读本是：W. Feller，*An Introduction to Probability Theory and Its Applications*，Vol. 1，second edition，Wiley-Interscience，New York，1957.

第 9 章　表格和信息检索

这一章继续学习自第 7 章开始的信息检索，但现在是集中在表格上而不是表上。我们从普通的矩形数组开始，然后考虑其他种类的数组，随后推广到哈希(hash，又称散列)表的学习。这一章的一个主要目的是分析和比较各种算法，了解不同条件下哪些算法更可取。这一章的应用包括一种基于表格的排序方法和使用哈希表的 Life 游戏。

9.1　引言：突破 lg *n* 的障碍

在第 7 章我们表明，单独使用键比较，平均起来不可能用少于 lg *n* 次比较去完成 *n* 个项的查找，但这个结果仅是针对用键比较的查找而言。如果我们能使用某些其他方法，则可以对数据进行安排，从而可以更快地定位一个给定的项。

*** 表格查找

实际上，我们通常都会这样做。如果有 500 个不同的记录，每一个都赋有 0~499 的一个索引，那么我们就决不会想到使用顺序查找或二分查找去定位一个记录，而只是将这些记录存储在一个大小为 500 的数组中，并通过普通的表格查找用下标 *n* 去定位第 *n* 项。

*** 信息检索函数

表格查找和查找享有同样的本质目的，即信息检索。我们从一个键开始(这个键可以很复杂，或者只是一个下标)，希望找到具有这个键的元素(如果有的话)的位置。换言之，表格查找和我们的查找算法提供了从键的集合到表或数组中的位置的函数。由于我们假设每个元素只有一个键，这个函数实际上是从实际出现的键的集合到实际出现的位置的集合的一一对一的映射，并且只有一个元素具有给定的键。

*** 表格

这一章将学习在顺序存储中实现表格和访问表格的方法，我们从普通的矩形数组开始，并考虑具有非零元素的受限位置的表格，如三角表格。然后转向更一般的问题，目的首先是引入和推动访问数组的使用，然后是哈希表在信息检索中的应用。

我们将看到，根据表格的形态，需要用几个步骤去检索一个元素，但即便如此，所需的时间仍然是 $O(1)$——即它受一个不依赖于表格长度的常数的约束——因此表格的查找可能比任何查找方法更有效。

*** 表格的下标

我们所考虑的表格元素将由一个整数序列索引，就像用这种序列索引数组一样，实际上，我们将用数组实现抽象定义的表格。为了区分抽象概念和它的实现，我们引入下面的符号：

> 约定
>
> 界定抽象定义的表格的元素下标括在圆括号中，
>
> 而数组元素的下标括在方括号中。

＊ ＊ ＊例子

因此 $T(1,2,3)$表示下标为序列 1,2,3 的表格 T 的元素，而 A[1][2][3]表示C++数组 A 的相应下标的元素。

9.2　矩形表格

由于矩形表格的重要性，几乎所有的高级语言都提供了方便有效的二维数组去存储和访问它们，因此通常程序设计者不必担心实现细节。尽管如此，表格在计算机中的存储基本上是按顺序序列安排(即以直线方式，一个接着一个)，因此对矩形表格的每次访问，机器必须做一些工作将矩形中的位置转换成直线上的位置。我们更仔细地来看一下这个过程。

1. 以行序为主序和以列序为主序

读取矩形表格的自然方法是从左到右读出第一行的元素，再读第二行的元素，如此一直到读完最后一行，这也是大多数编译器存储矩形数组的次序，这种方式称为以行序为主序(row-major ordering)。例如，假设表格的行是从 1 到 2 编号，列是从 5 到 7 编号(由于我们是在通用表格上工作，因此没有理由必须受 C 和C++要求的所有数组的下标都从 0 开始的限制。)，以行序为主序存储的元素在表格中的下标次序是：

(1,5) (1,6) (1,7) (2,5) (2,6) (2,7)

＊ ＊ ＊Fortran：以列序为主序

以行序为主序是C++和大多数高级语言中存储二维数组元素所使用的次序，而标准的 Fortran 语言使用以列序为主序，先是第一列的元素，然后是其他列。本例以列序为主序的存储为：

(1,5) (2,5) (1,6) (2,6) (1,7) (2,7)

图 9.1 进一步说明了以行序为主序和以列序为主序的 3 行 4 列的表格。

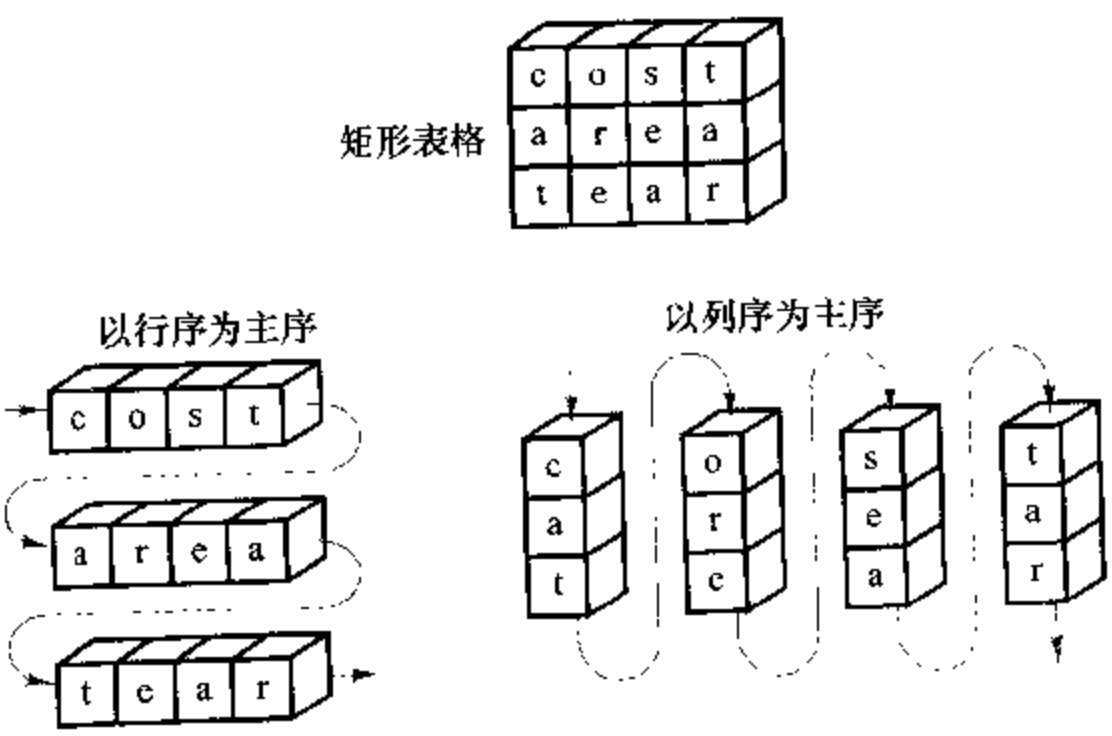


图 9.1　矩形数组的顺序表示

2. 建立矩形表格的下标

＊ ＊ ＊矩形数组与下标函数

在一般问题中，编译器必须能够从下标(i,j)开始，计算表格的对应元素在顺序数组中存储的位置。我们将为这种计算推导出一个表达公式。为简单考虑，我们仅以行序为主序，并假设行是从0到$m-1$编号，列是从0到$n-1$编号，它的一般情况则留作一个练习。这个表格总共有mn个元素，它在数组中的顺序实现也必须是一样多的元素。为了得到计算(i,j)的位置的公式，首先考虑一些特殊情况。显然$(0,0)$是在位置0，而且整个第一行都很简单：$(0,j)$就是位置j。第二行的第一个元素$(1,0)$是在$(0,n-1)$之后，因此是在位置n。继续下去，我们发现$(1,j)$是在位置$n+j$。下一行的元素之前将有两个满行(即$2n$个元素)，因此元素$(2,j)$是在位置$2n+j$。概括地说，第i行之前有ni个元素，因此我们想要的公式是：

矩形表格中的元素(i,j)在顺序数组中位于第$ni+j$个位置上。

这个公式给出了表格元素的顺序位置，我们称它为下标函数。

3. 变化：访问数组

矩形表格的下标函数当然不难计算，大多数高级语言的编译器在矩形数组每次被引用时，将此计算的必要步骤写成机器语言程序。然而在较小的机器上，乘法可能相对缓慢，因此可使用稍微不同的方法去消除乘法。

＊ ＊ ＊矩形表格与访问数组

这个方法是保留一个辅助数组，它是n的乘法表的一部分，这个数组包含这些值：$0,n,2n,3n,\cdots,(m-1)n$。注意，这个数组(一般)比矩形表格更小，因此它能永久驻留在内存中而不损失太多空间。它的元素仅需要计算一次(并且注意它们能够只用加法计算出来)，对矩形表格以后所有的引用，编译器通过取出辅助表格中位置i的元素再加上j，找到(i,j)的位置并访问结果位置。

这个辅助数组为我们提供了访问数组(access array，见图9.2)的第一个例子。一般来说，访问数组是一个辅助数组，用于找到存储在其他位置的数据。访问数组有时也称为访问向量。

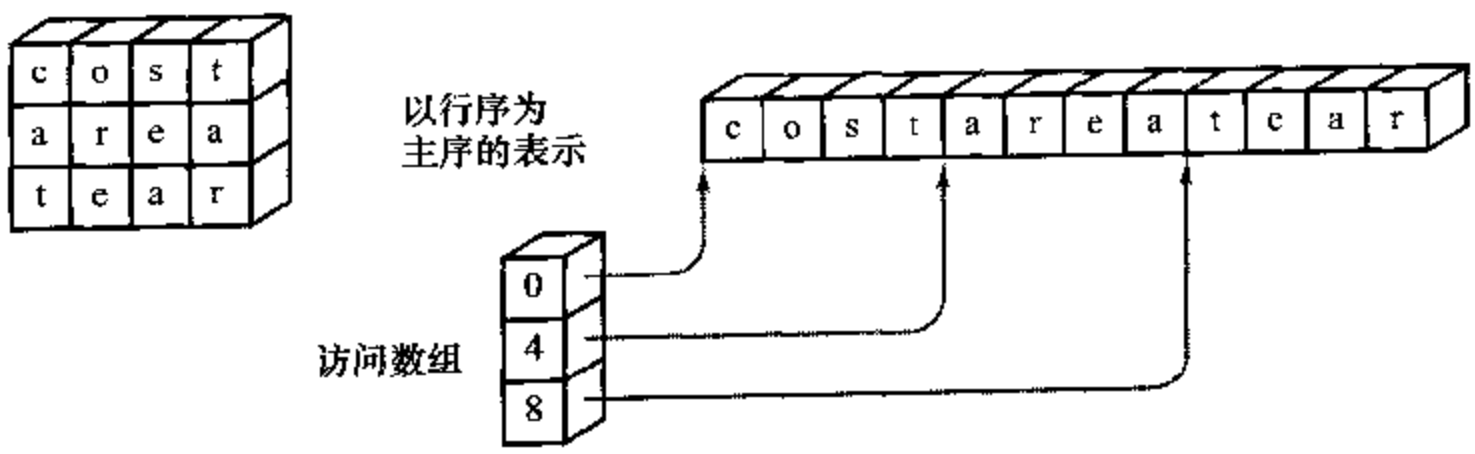


图9.2 矩形表格的访问数组

练习 9.2

E1. 一个二维矩形表格，它的行下标从 0 到 $m-1$，列下标从 0 到 $n-1$，其中包含 0，$m-1$ 和 $n-1$，那么在以列序为主序的情况下，它的下标函数是什么？

E2. 对一个行下标是任意边界 $r\sim s$（含 r 和 s），列下标从 $t\sim u$（含 t 和 u）的二维表格，以行序为主序给出下标函数。

E3. 对一个 d 维且每维都具有任意边界的表格，用广义的以行序为主序找出其下标函数。

9.3　各种形态的表格

* * * 矩阵

通常，存储在矩形表格中的信息表示不需要用到矩形中的每个位置。如果将矩阵定义为数值的矩形表格，那么矩阵中的一些位置经常要求是 0，图 9.3 中给出了这样的几个例子。即使表格中的元素不是数值，实际使用的位置可能也不是矩阵中的全部位置，可能存在比使用矩形数组并让一些位置空着更好的实现方法。这一节，我们考查实现各种形态表格的方法，它们不要求留出矩形数组中未用的空间。

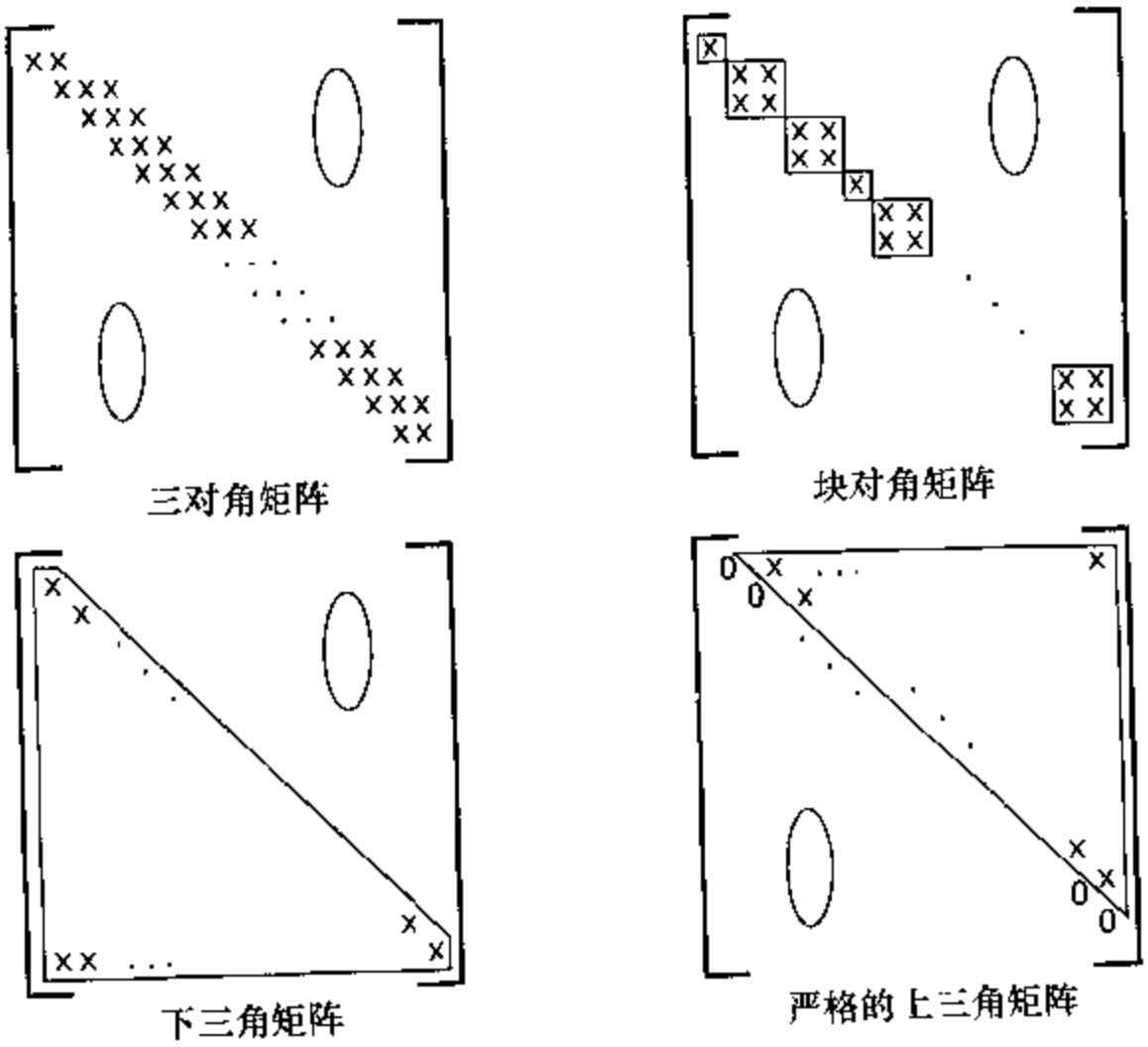


图 9.3　各种形态的矩阵

9.3.1 三角表格

我们考虑图 9.3 中显示的下三角表格的表示法，这种表格正式地定义为所有下标(i,j)都要求满足 $i \geqslant j$ 的表格。我们可以在顺序数组中实现三角表格。如图 9.4 所示，将每一行在其上一行后面滑出。

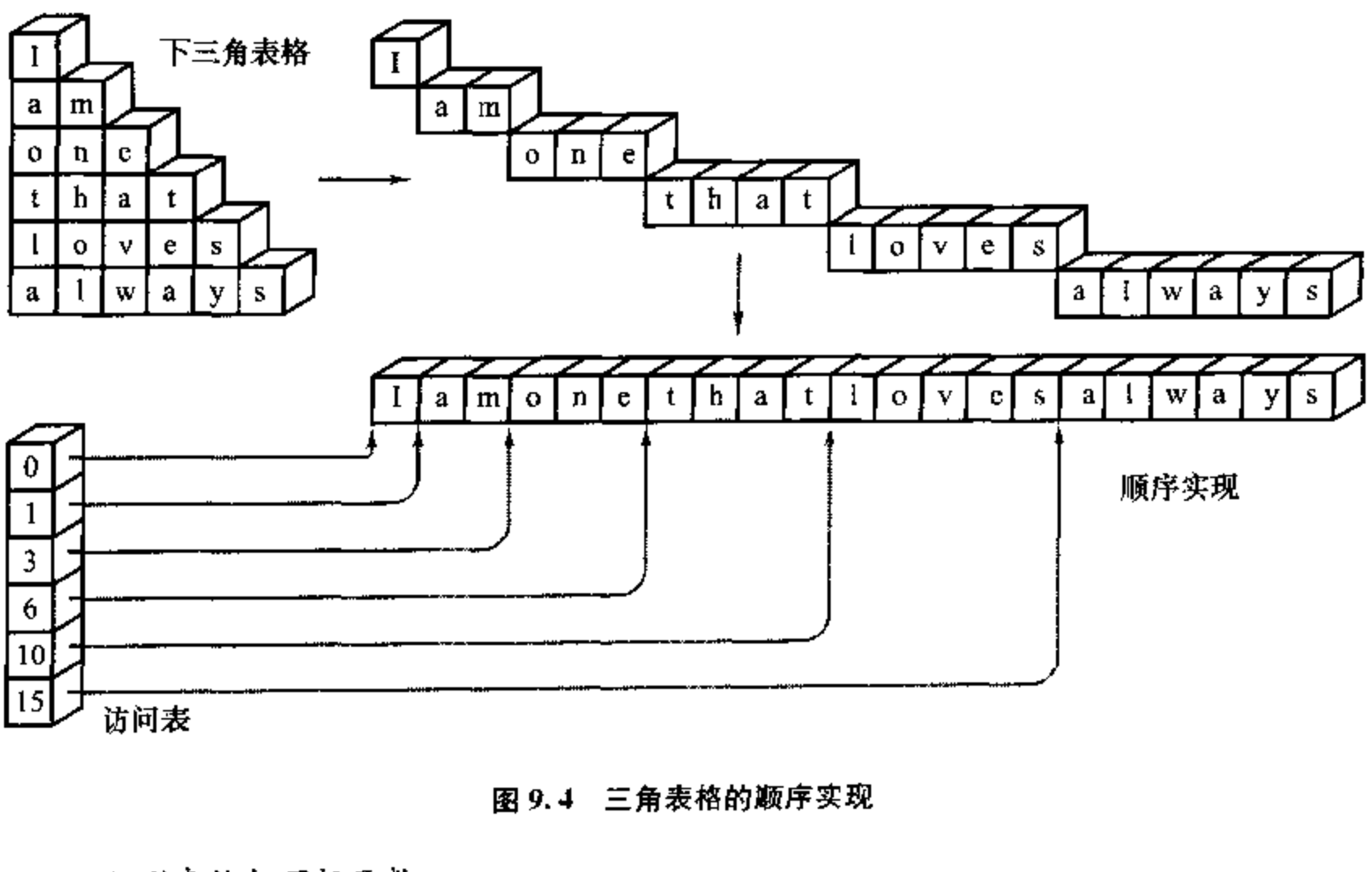


图 9.4 三角表格的顺序实现

* * * 矩形表格与下标函数

为了构造描述这种映射关系的下标函数，我们再做一点简化，假设行和列从 0 开始编号。为找出(i,j)对应的位置，必须找到第 i 行开始的位置，然后要定位第 j 列时只需将 j 加到第 i 行的起始位置。如果顺序数组的元素也从 0 开始编号，则起始点的下标将与第 i 行前的元素的编号一样。显然在第 0 行前有 0 个元素，在第 1 行前只有第 0 行的 1 个元素，第二行前有 $1+2=3$ 个元素，通常第 i 行前正好有 $1+2+\cdots+i=\frac{1}{2}i(i+1)$ 个元素。[①] 因此期望的函数是：三角表格的元素(i,j)对应到顺序数组的元素 $\frac{1}{2}i(i+1)+j$。

* * * 三角表格与访问数组

就像我们对矩形数组的做法，通过设立一个访问数组，使它的元素对应到三角表格的行下标，我们能够避免所有的乘、除法。访问数组的位置 i 将永久地包含数值 $\frac{1}{2}i(i+1)$。这个访问数组只需要在程序启动时计算一次，然后可在每次引用三角表格时重复使用。注意，甚

① 见附录 A 对此等式的证明。

至这个访问数组的初始计算也不需要乘法或除法，只要用加法按次序计算这些元素：

$$0,1,1+2,(1+2)+3,\cdots$$

9.3.2　锯齿表格

在前面的两个例子中，我们都是将矩形表格看成是由它的行组成。在普通的矩形数组中，所有这些行都有相同的长度；在三角表格中，每行的长度可从一个简单的公式推出。我们现在考虑如图 9.5 中所示的锯齿表格的情况，这种表格在行的位置和其长度之间没有可预知的关系。

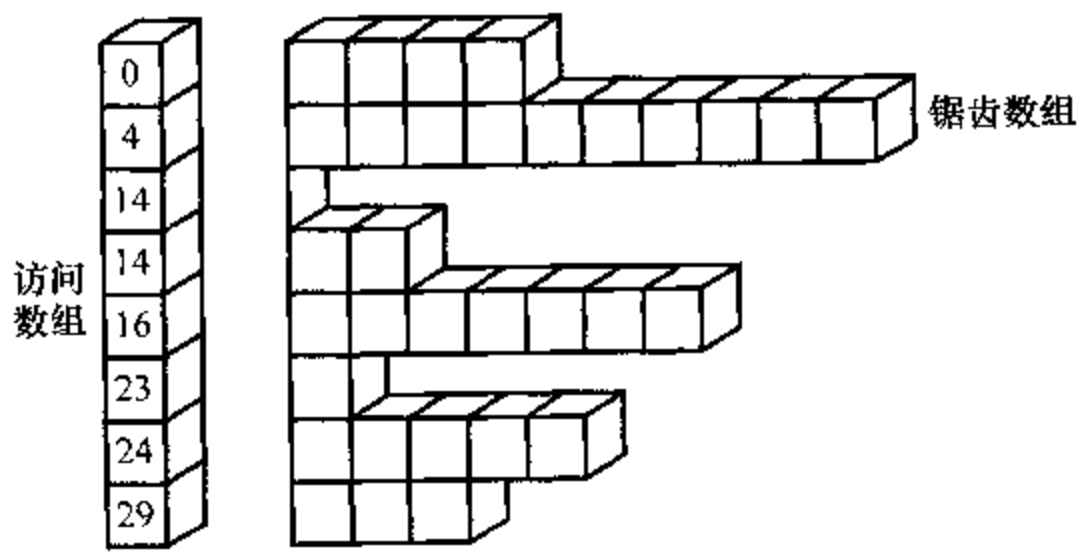


图 9.5　锯齿表格的访问数组

从这个图中显然可以看出，即使我们不能给出将锯齿数组映射到顺序存储的先验函数，但使用访问数组仍可以使这种映射变得像前面的例子那样简单，而且能尽快地引用锯齿数组的元素。为设立这个访问数组，我们必须从它的第一行开始，按锯齿表格的自然顺序去构造它。与以前一样，访问数组的第 0 个元素是顺序数组的开始。在构造了锯齿表格的每一行后，然后顺序存储的第一个未用位置的下标作为访问数组中的下一个元素，并用于开始构造锯齿表格的下一行。

9.3.3　反转表格

接下来我们考虑一个例子，它说明多个访问数组，利用这些数组，我们一次能够以几个不同的键引用单个的记录表格。

* * * 多重记录

考虑电话公司在访问其客户记录时面临的问题。为了出版电话簿，这些记录必须按用户姓名的字母顺序排序，但为了处理长途话费，这些账号必须按电话号码排序，而为了例行维护，公司也需要将用户按地址排序，从而修理工能够一次处理几条线路。令人信服的是，电话公司能保存 3 个(或更多的)记录集合，其中一种按姓名排序，一种按号码排序，另一种按地址排序。然而这种方式不仅非常浪费存储空间，而且将引入无尽的令人头痛的问题，如果一个集合中的数据修改了，而另一个却没有，那么就可能使用错误的和无法预知的信息。

* * * 多重访问数组

通过使用访问数组，我们能够避免记录的多重集合，并且仍然能用这 3 个键中的任一个查找记录，几乎就像查找按那个键完全排序的记录一样快。对姓名，我们设立一个访问数组，表格中的第一个元素是姓名按字母顺序存储的第一个用户记录的位置，第二个元素是第二个(按字母顺序)用户记录的位置，以此类推。第二个访问数组的第一个元素恰好是按数字次序存储的电话号码最小的那个用户记录的位置。而在第三个访问数组中，元素给出了按地址的字典次序排序后的记录位置。

图 9.6 用少数账号举例说明了这种方案。

索引	姓名	地址	电话
1	Hill, Thomas M.	High Towers #317	2829478
2	Baker, John S.	17 King Street	2884285
3	Roberts, L. B.	53 Ash Street	4372296
4	King, Barbara	High Towers #802	2863386
5	Hill, Thomas M.	39 Kill Street	2495723
6	Byers, Carolyn	118 Maple Street	4394231
7	Moody, C. L.	High Towers #210	2822214

访问数组		
姓名	地址	电话
2	3	5
6	7	7
1	1	1
5	4	4
4	2	2
7	5	3
3	6	6

图 9.6 多重键访问数组：反转表格

* * * 用于有序访问数组的无序记录

注意，在这种方法中，作为键对待的所有成员都以同样的方式处理。对记录本身需要按一个键而不是另一个键排序，或者实际上为何它们需要完全排序，不存在特别的理由。记录本身可以按任意顺序保存——也就是说，按它们首次输入系统的顺序。记录是保存在一个数组中，而访问数组中的元素是此数组的下标，或者还是保存在动态内存中，访问数组保存指向各个记录的指针，这之间并无多大区别。在任何情况下，是访问数组用于信息检索，而且与普通的顺序数组一样，它们可用于表格查找、二分查找或适合顺序实现的任何其他目的。

练习 9.3

E1. 方阵的主对角线由具有相同行、列坐标的元素组成。对角矩阵是一种方阵，它所有的非主对角线元素均为 0。描述一种存储对角矩阵的方法，它对必定为 0 的元素不用花费空间，然后给出相应的下标函数。

E2. 三对角矩阵是一种方阵，除了主对角线以及直接在主对角线上下的元素之外，其他所有元素都是 0。即，T 是一个三对角矩阵意味着 $T(i,j)=0$，除非 $|i-j|\leqslant 1$。

(1)为三对角矩阵设计一种节省空间的存储方案，并给出相应的下标函数。

(2)矩阵的转置是通过互换矩阵的行和列而得到的新矩阵。即矩阵 B 是矩阵 A 的转置意味着对矩阵中所有的位置 i 和 j，有 $B(j,i)=A(i,j)$。使用此练习前一部分设计的存储方案，设计一个转置三对角矩阵的算法。

E3. 上三角矩阵是一种方阵，所有主对角线之下的元素都为 0。描述用访问数组方法存储上三角矩阵所必需的修改。

E4. 考虑图 9.7 中显示的三角形状表格，它的列下标是从 $-n$ 到 n，行下标是从 0 到 n。

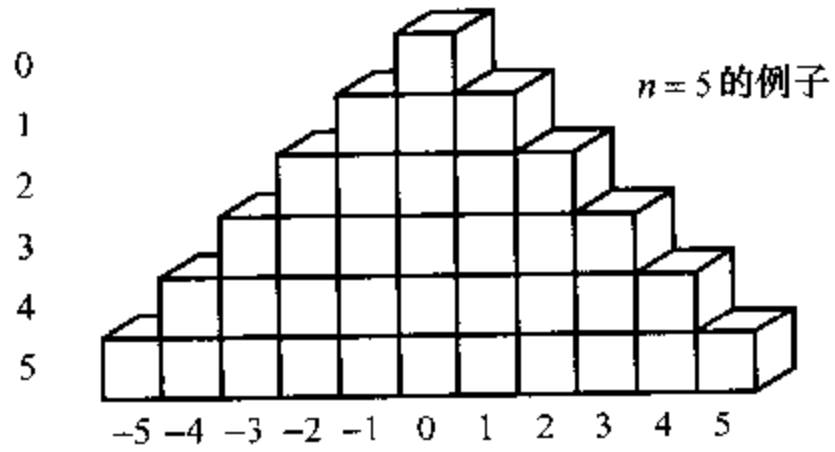


图 9.7　关于 0 列对称的三角表格

(1)设计一个下标函数，它将这种形式的表格映射到一个顺序数组中。

(2)编写函数生成访问数组，用于在顺序数组中查找这个表格每一行的第一个元素。

(3)编写函数用于从左到右反射这个表格，其中 0 列(中央一列)的元素保持不动，在 -1 列的元素与 1 列的元素互相交换，以此类推。

程序设计项目 9.3

实现课本中描述的使用访问数组存储下三角表格的方法，并在下列设计项目中加以应用。

P1. 编写函数从终端读入下三角表格的元素，元素的类型应当为 double 型。

P2. 编写函数在终端上输出一个下三角表格。

P3. 假设下三角表格就像在地图中经常出现的城市间距离的表格。编写函数检查三角规则：从城市 A 到城市 C 的距离决不超过从城市 A 到城市 B 的距离加上从城市 B 到城市 C 的距离。

P4. 将前几个项目中的函数嵌入到一个完整的程序中，用于演示下三角表格。

9.4　表格：一种新的抽象数据类型

在本章开始处，我们研究了几种用于查找表格中元素的下标函数，然后我们转向与下标函数具有同样使用目的的访问数组。函数和表格查找之间确实非常类似：对于函数，我们从自变量开始并计算出相应的值；对于表格，我们从下标开始，并查找相应的值。我们现在使

用这种类似性去得到“表格”这个术语的正式定义，这个定义反过来又会激发在下一节中完成新的思想。

1. 函数

＊＊＊定义域、上域和范围

在数学中，利用两个集合和从第一个集合的元素到第二个集合的元素的对应去定义函数。如果 f 是从集合 A 到集合 B 的一个函数，则 f 给 A 的每个元素分配 B 的惟一元素，集合 A 称为 f 的定义域(domain)，集合 B 称为 f 的上域(codomain)。仅包含作为 f 的值出现的那些元素的 B 的子集称为 f 的范围(range)。图 9.8 中用图例说明了这个定义。

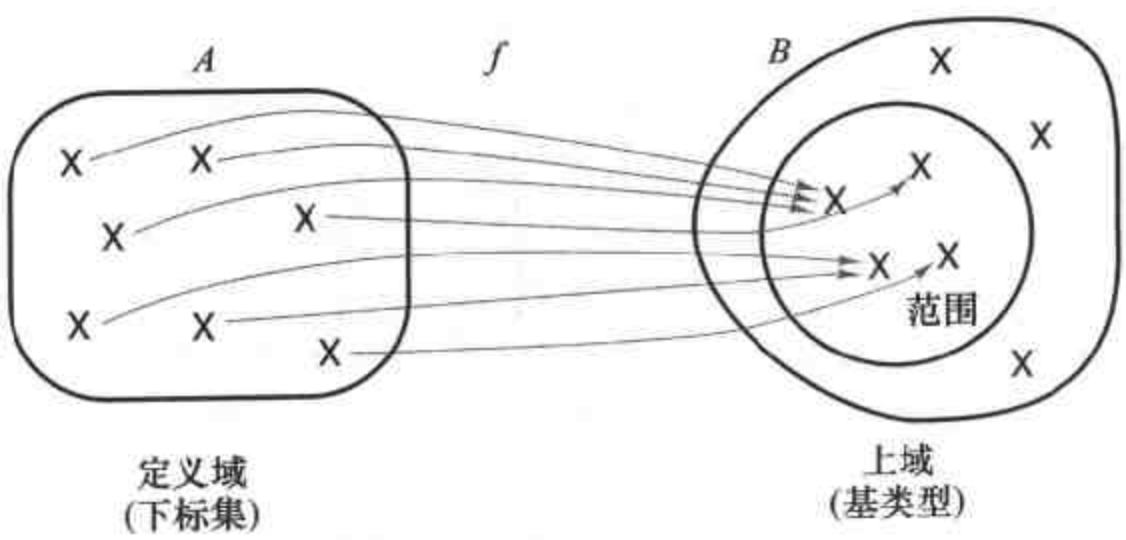


图 9.8　函数的定义域、上域和范围

＊＊＊下标集、值类型

表格访问从一个索引开始，使用此表格查找相应的值，因此对表格来说，我们称定义域为下标集(index set)，并称上域为基类型(base type)或值类型(value type)(回忆 4.6 节中曾将类型定义为值的集合)。例如，如果我们有数组声明 **double** array[n];，那么下标集是 0 到 $n-1$ 之间的整数集合，基类型是所有实数集合。作为第二个例子，考虑具有 m 行的三角表格，它的元素具有 Item 类型，那么它的基类型只是 Item 类型，而其索引类型是有序整数对的集合 $\{(i,j) \mid 0 \leqslant j \leqslant i < m\}$。

2. 一种抽象数据类型

我们现在准备将表定义为一种新的抽象数据类型，但回忆 4.6 节中完成的定义，我们也必须指定能执行的操作。在此之前，我们概括一下已学的知识。

＊＊＊ADT 表格

定义　具有下标集 I 和基类型 T 的表格是从 I 到 T 的一个函数，同时包含下列操作。
(1)表格访问：在 I 中的任何索引处评价此函数。
(2)表格赋值：通过将 I 中指定下标处的值修改成赋值中指定的新值去修改函数。

这两个操作是C++和某些其他语言中为数组提供的所有操作，但没有理由不让我们允许自己的抽象表格有更多的操作存在。如果比较表的定义，可以发现除了访问和赋值以外，我们还允许插入和删除，因此对表格也可能做同样的处理。

(3)创建:建立一个从 I 到 T 的新函数。
(4)清除:从下标集 I 中移去所有元素,因此剩下的定义域是空的。
(5)插入:将新元素 x 附加到下标集 I 中并定义此函数在 x 上的相应的值。
(6)删除:从下标集 I 中删除元素 x 并将函数限制到结果生成的较小的定义域中。

即使C++中的数组不能直接利用后面的这些操作,但它们对许多应用仍十分有用,因此我们将在下一节进一步研究它们。其他一些语言如 APL 和 Snobol,在程序运行期间改变表格的规模是一个重要的特性。在任何情况中,我们总是应该仔细地用一门语言进行程序设计,而且决不能让自己的想法被特定语言的局限性所限制。

3. 实现

*** 下标函数和访问数组

*** 分而治之

刚才给出的定义是一种抽象数据类型,它本身并不涉及实现,也没有谈及前面学习的下标函数或访问数组。下标函数和数组实际上是更通用表格的实现方法。下标函数或访问数组从某种指定形式的通用下标集开始,并作为结果生成某个下标范围的下标,如整数的子区间,这个范围然后能直接由程序设计语言用作数组的下标。这样,表格的实现就分成两个更小的问题:找到一个访问数组或下标函数和用一个数组进行程序设计。应当注意,这两个问题都是表格的特殊情况,因此我们就有了将问题分成具有同样性质的两个更小的问题再去求解它的一个例子,图 9.9 中说明了这个过程。

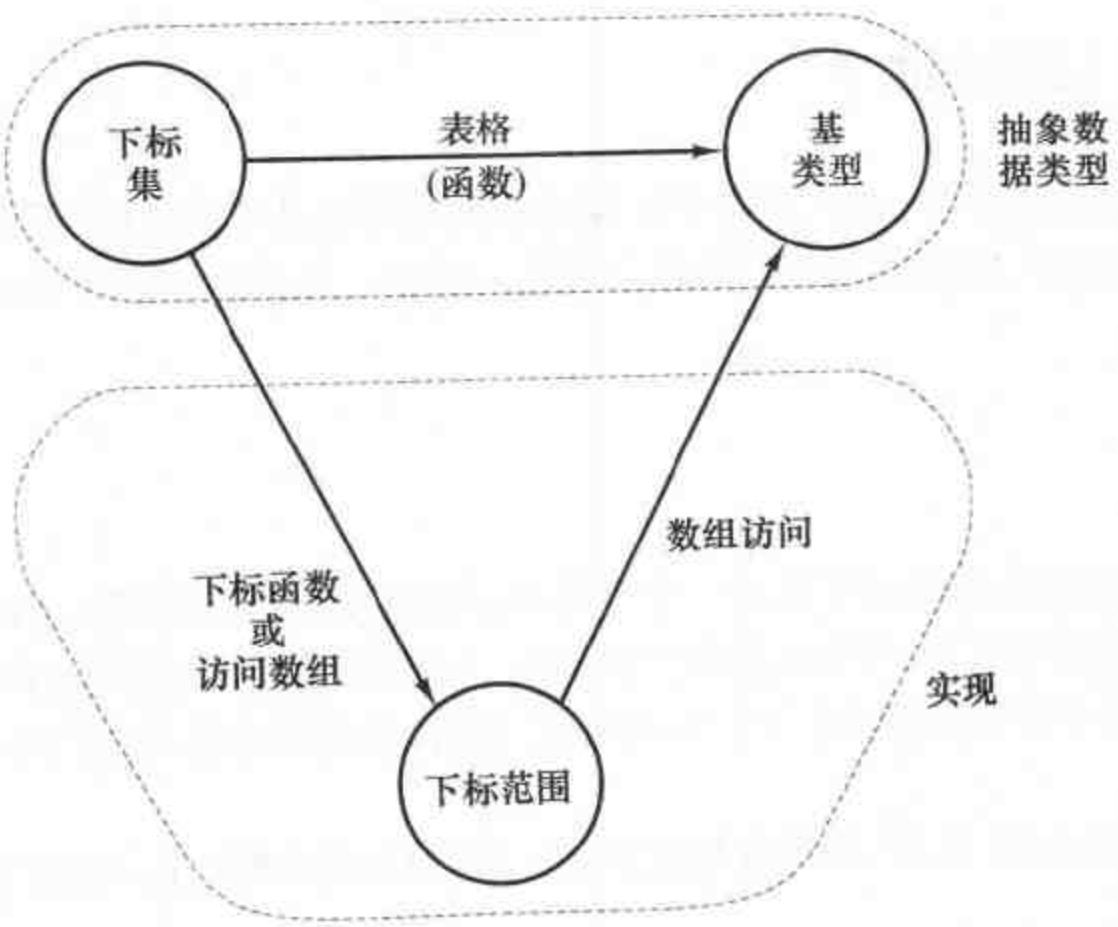


图 9.9　表格的实现

4. 比较

* * * 表和表格

× × × 检索

我们比较抽象数据类型表(list)和表格,表潜在的数学结构是序列,而表格是集合和函数。序列有隐含的次序,第一个元素,第二个元素,以此类推;但集合和函数没有这样的次序(如果下标集有某种自然次序,则有时这种次序会反映到表格中,但这不是使用表格的一个必要方面)。因此表的信息检索自然包含如前些章中学习的查找,但表格的信息检索要求不同的方法,这些访问方法直接定位到想要的元素。查找表所需要的时间通常依赖于表中元素的个数 n,而且至少是 $\lg n$(见定理 7.6),但访问表格的时间通常不依赖于表格中元素的个数,也就是说,它通常是 $O(1)$。为此,在许多应用中,表格访问明显快于表查找。

* * * 遍历

另一方面,遍历是表的一种自然操作,但对表格来说并不是。通常易于在表中从头到尾移动以对表中的每个元素执行某种操作,而对表格中的每个元素执行一个操作则并不是这样简单,特别是当元素事先指定了某种特殊次序时。

* * * 表格和数组

最后,我们将澄清表格和数组这两个术语之间的区别。一般说来,我们使用的表格正如本节定义的,并将数组这个术语限于表示在C++和大多数高级语言中可用且用于实现表格和顺序表的程序设计特征。

9.5 应用:基数排序

在计算机之前出现的一种正式的排序算法最初是为打孔卡设计的,但能够将它设计成一种非常有效的排序方法,用于使用表格和队列的链表。此种算法应用于使用字符串对象作为键的记录。

9.5.1 思想

这种方法的思想是每次考虑键的一个字符并分割这些元素,不是将它们分成两个子表,而是尽键中给定字符的可能性分成许多子表。例如,如果键是单词或其他字母串,则将表在每个阶段分成 26 个子表,即创建一个 26 个表的表格,并按键中字符的大小将元素分配到表中。

旧模式的打孔卡有 12 行,因此将机械卡排序器设计成每次仅在一列上工作,并且将这些卡分成 12 堆。

用这种方法对单词排序的人可能会首先按起始字母将单词分配到 26 个表中(或将打孔卡分成 12 堆),然后将其中的每个子表按照第二个字母分成更多的子表,以此类推。下面的思想消除了子表的这种多重性:首先根据最低有效位(least significant position)而不是最高有效位,将这些项分布到子表的表格。在这第一次划分后,表格中的表按照最低有效位中的

字符给定的次序被一起作为单个的表收回。然后按次最低有效位划分这个表格并重新组合成一个表。重复这些步骤，直到在最高有效位上划分这个表并重新组合它，这个表就是完全有序的。

* * * 例子

图 9.10 中通过对 9 个 3 字母的单词表进行排序说明了这个过程，左边一列是以初始次序显示的这些单词。第一次是按照第 3 个字母将它们分成 3 个表，如第 2 列中所示，其中浅灰色框指示结果生成的子表，每个子表中的单词保持与划分前同样的相对次序。接下来，这些子表如图的第 2 列所示被一起收回，并按第 2 个字母将它们分成 2 个子表。最后，这些子表又被重新组合并按第 1 个字母分配到 4 个子表中，当这些子表被重新组合后，整个表就是有序的。

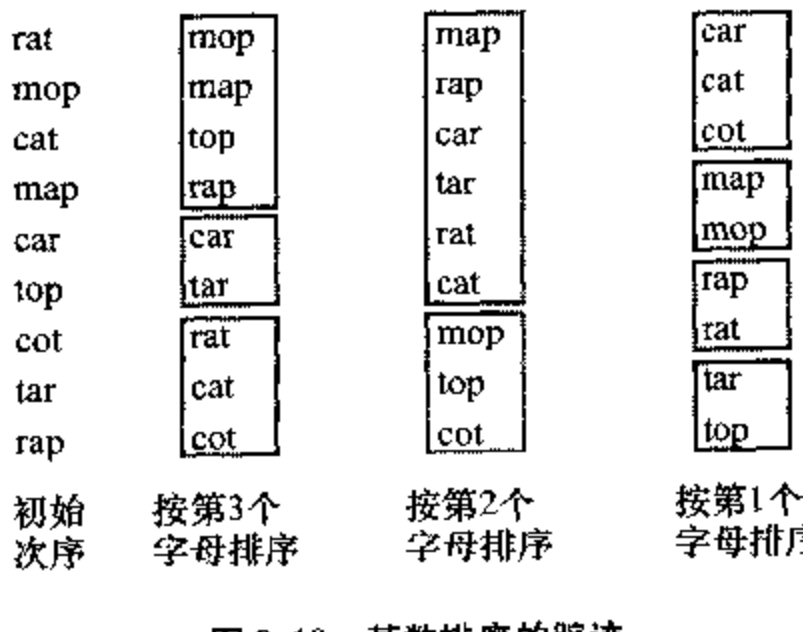


图 9.10　基数排序的踪迹

9.5.2　实现

我们用C++去实现键是字母数字串的记录表的这种方法。在这些项每次被划分成表格的子表后，这些子表必须重新组合成单个的表，使得这些项能够按键的下一个最高有效位重新分配。由于元素总是插在子表末端，而重组子表时又是从开始处移走，因此我们将子表作为队列处理。

为表达清晰，我们对这个方法将使用自己开发的通用表和队列软件包。然而这样做会负担一些不必要的数据移动。例如，如果我们采用恰当实现的链表和链式队列，则仅通过将每个队列的尾部连接到下一个队列的头部，就能重新将链式队列组合成一个表，图 9.11 中利用前面用过的同样的 9 个单词的表说明了这个过程。在每个阶段，以黑色显示的链是那些在某一个队列中的链，以浅灰色显示的链是为了将队列重组为单个表而加入的链。对基数排序的这种优化进行设计需要一个提供连接的派生链表类的实现，我们将此留作一个项目设计。

为了允许缺少的或非字母的字符，需要创建一个有 max_chars=28 个队列的数组，位置 0 对应于空格符，位置 1～26 对应于字母(将大、小写字母看作同样的)，位置 27 对应于键中出现的任意其他字符。在从键的最低有效位到最高有效位的循环中，我们将遍历这个链表，并将每一项加到适宜队列的尾部。在对这个表如此分割后，我们又将队列重新组合成一个

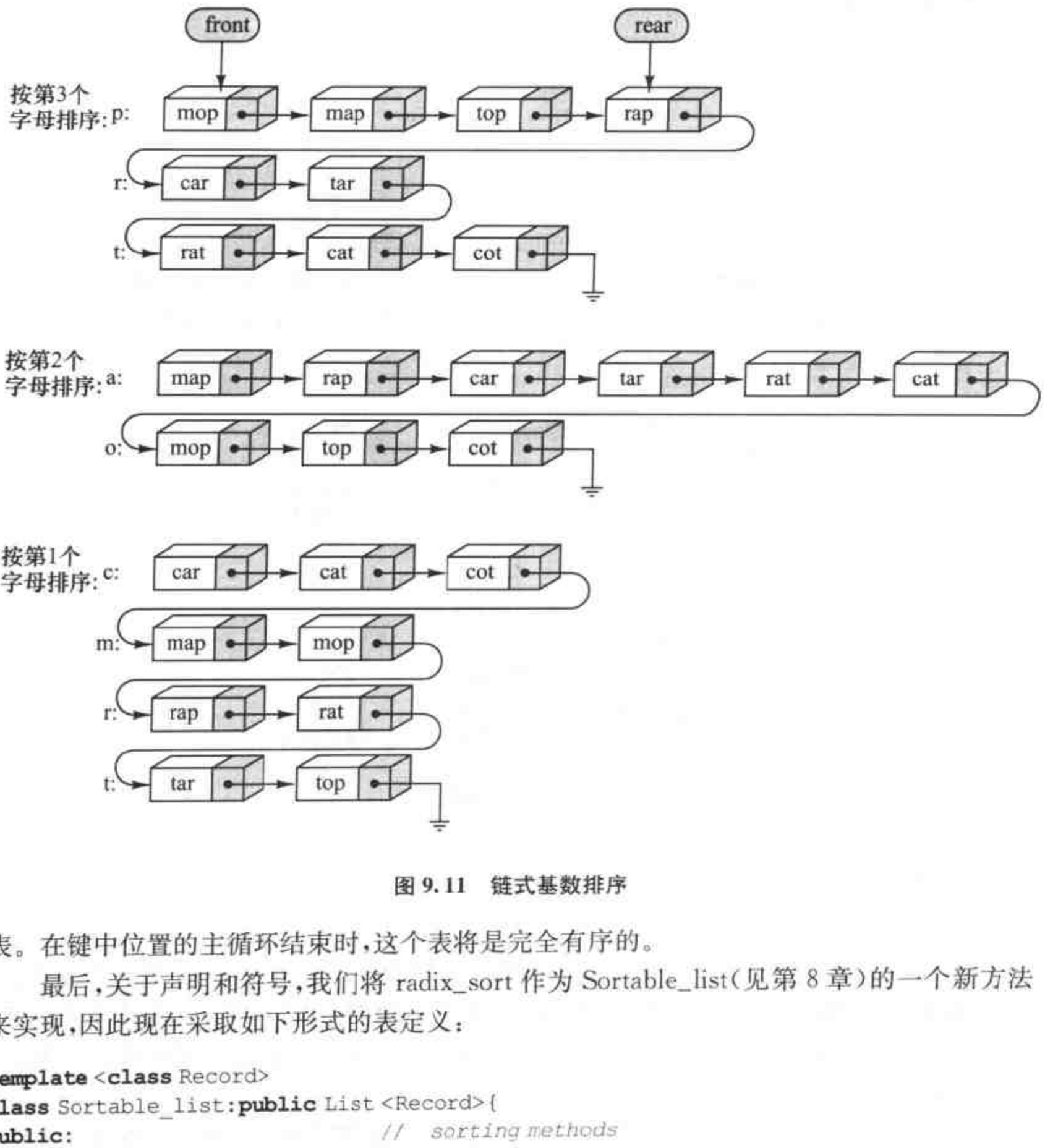


图 9.11 链式基数排序

表。在键中位置的主循环结束时,这个表将是完全有序的。

最后,关于声明和符号,我们将 radix_sort 作为 Sortable_list(见第 8 章)的一个新方法来实现,因此现在采取如下形式的表定义:

```
template <class Record>
class Sortable_list:public List <Record>{
public:                              //  sorting methods
  void radix_sort();
  //  Specify any other sorting methods here.
private:                             //  auxiliary functions
  void rethread(Queue queues[]);
};
```

这里基类 List 可以是第 6 章中学过的任何一种实现,辅助函数 rethread 将用于重组队列。

Record 类的需求类似于第 8 章中的用法,但此处的每个 Record 使用字母数字串作为它的键。我们将使用 Record 的方法 char key_letter(int position)返回键的特定位置中的字符(或当键的长度小于位置时返回一个空格),因此 Record 的定义基于下面的框架:

```
class Record{
public:
```

```
  char key_letter(int position)const;
  Record();                          //  default constructor
  operator Key()const;               //  cast to Key
//  Add other methods and data members for the class.
};
```

1. 排序方法

排序方法采用下面的形式：

```
const int max_chars=28;
template <class Record>
void Sortable_list<Record>::radix_sort()
/*Post: The entries of the Sortable_list have been sorted so all their keys are in
        alphabetical order.
  Uses: Methods from classes List, Queue, and Record;
        functions position and rethread.*/
{
  Record data;
  Queue queues[max_chars];
  for (int position=key_size-1;position>=0;position--){
    //  Loop from the least to the most significant position.
    while (remove(0,data)==success){
      int queue_number=alphabetic_order(data.key_letter(position));
      queues[queue number].append(data);   //  Queue operation.
    }
    rethread(queues);                      //  Reassemble the list.
  }
}
```

这个函数使用两个辅助子程序：alphabetic_order 用于确定哪个 Queue 对应到特定的字符，Sortable_list::rethread()用于将队列重组成重新有序的 Sortable_list。我们能利用第 3 章和第 4 章中 Queue ADT 的任何实现中的 Queue 操作。

2. 选择队列

函数 alphabetic_order 检查一个特定字符是否是字母并将它分配到合适的位置上，除空格以外的所有非字母字符都放到位置 27 上，空格符分配到位置 0。这个函数也需要做一些调整以不区分字母的大小写。

```
int alphabetic_order(char c)
/*Post: The function returns the alphabetic position of character c, or it returns 0 if
        the character is blank.*/
{
  if (c==' ')return 0;
  if ('a'<=c&&c<='z')return c-'a'+1;
  if ('A'<=c&&c<='Z')return c- 'A'+1;
  return 27;
}
```

3. 连接队列

函数 rethread 将 28 个队列重新连接成一个更新的 Sortable_list，并将这些队列清空以

使用于排序过程的下一次重复。本节后面的一个项目设计要求以依赖于实现的方式重新编写这个函数，使它比当前的版本运行得更快。

```
template <class Record>
void Sortable_list<Record> :: rethread(Queue queues[])
/*Post: All the queues are combined back to the Sortable_list, leaving all the queues
        empty.
  Uses: Methods of classes List, and Queue.*/
{
  Record data;
  for (int i=0;i<max_chars;i++)
    while (!queues[i].empty()){
      queues[i].retrieve(data);
      insert(size(), data);
      queues[i].serve();
    }
  }
```

9.5.3 分析

注意基数排序使用的时间是$\Theta(nk)$，其中n是待排序的项数，k是键中的字符数。我们设计的所有其他排序方法的运行时间依赖于n，但不直接依赖于键的长度，最佳时间来自归并排序，它是$n\lg n+O(n)$。

这些方法的相对性能因此在某种程度上与nk和$n\lg n$，即k和$\lg n$的相对大小有关。如果这些键较长但数目相对较少，则k较大且$\lg n$相对较小，其他的方法（如归并排序）会胜过基数排序，但如果k较小（键较短）且有大量的键，则基数排序会比我们学过的任何其他方法都要快。

练习 9.5

E1. 在用于跟踪其他排序方法的14个名字上跟踪基数排序的操作：

Tom Dot Eva Roy Tom Kim Guy Amy Jon Ann Jim Kay Ron Jan

E2. 将下面7个数看作两位数字的整数，在这7个数的表上跟踪基数排序的操作：

26 33 35 29 19 12 22

E3. 将前面7个数字各看作6位二进制整数，在这样的表上跟踪基数排序的操作。

程序设计项目 9.5

P1. 设计、编程并测试一个独立于实现的字母键上的基数排序版本。

P2. 课本中提供的基数排序程序效率较低，这是由于其独立于实现的特性必须进行大量的数据移动。设计一个项目，它依赖于实现且节省所有的数据移动。在 rethread 中仅需要将队列的队尾链到下一个队列的队头，这种链接操作要求访问受保护的 Queue 数据成员。换言之，我们需要一个修改的 Queue 类，实现它的一种简单方法是向我们的派生

链表实现中加入一个方法 Sortable_list::(**const** Sortable_list &add_on);，并在基数排序的代码中使用表而不是队列。将这个版本的性能与链表的其他排序方法的性能进行比较。

9.6　哈希法

9.6.1　稀疏表格

1. 下标函数

即使当键不再如数组中直接用作索引的下标，我们仍能继续设计表格查找。我们所能做的是在用于检索信息的键和用于访问数组的下标之间建立一一对应的关系。这时生成的下标函数会比前几节中的那些函数更加复杂，因为它需要将键从字母信息转换成整数，但在原理上它仍然是能够实现的。

产生的惟一困难是可能的键数超过了表格可用的空间量。例如，如果键是 8 个字母构成的单词，那么就有 $26^8 \approx 2\times10^{11}$ 个可能的键，这个数可能大于高速内存中可用的位置数。然而实际上，只有其中一小部分键会真正出现。也就是说，这个表格是稀疏的(sparse)。我们在概念上可以认为它是由一个非常大的集合编入索引，但实际上只会占据相对少的位置。抽象地看，我们可以按照如下概念化的声明进行思考：**class** … {**private**: sparse table(Key) of Record;};。即使不可能直接实现如此的说明，但如果从这样的描述出发，而对有关如何实现的细节逐渐进行限制，则通常会有益于问题的求解。

2. 哈希表

＊＊＊下标函数不是一对一

哈希表(hash table，如图 9.12 中显示)的思想允许多个不同的键在下标函数的作用下映射到数组中同一位置。这样就存在两个记录会在同一位置的可能性，但如果真正出现的记录数相对于数组长度而言比较小，那么这种可能性不会引起时间的损失。即使当数组中

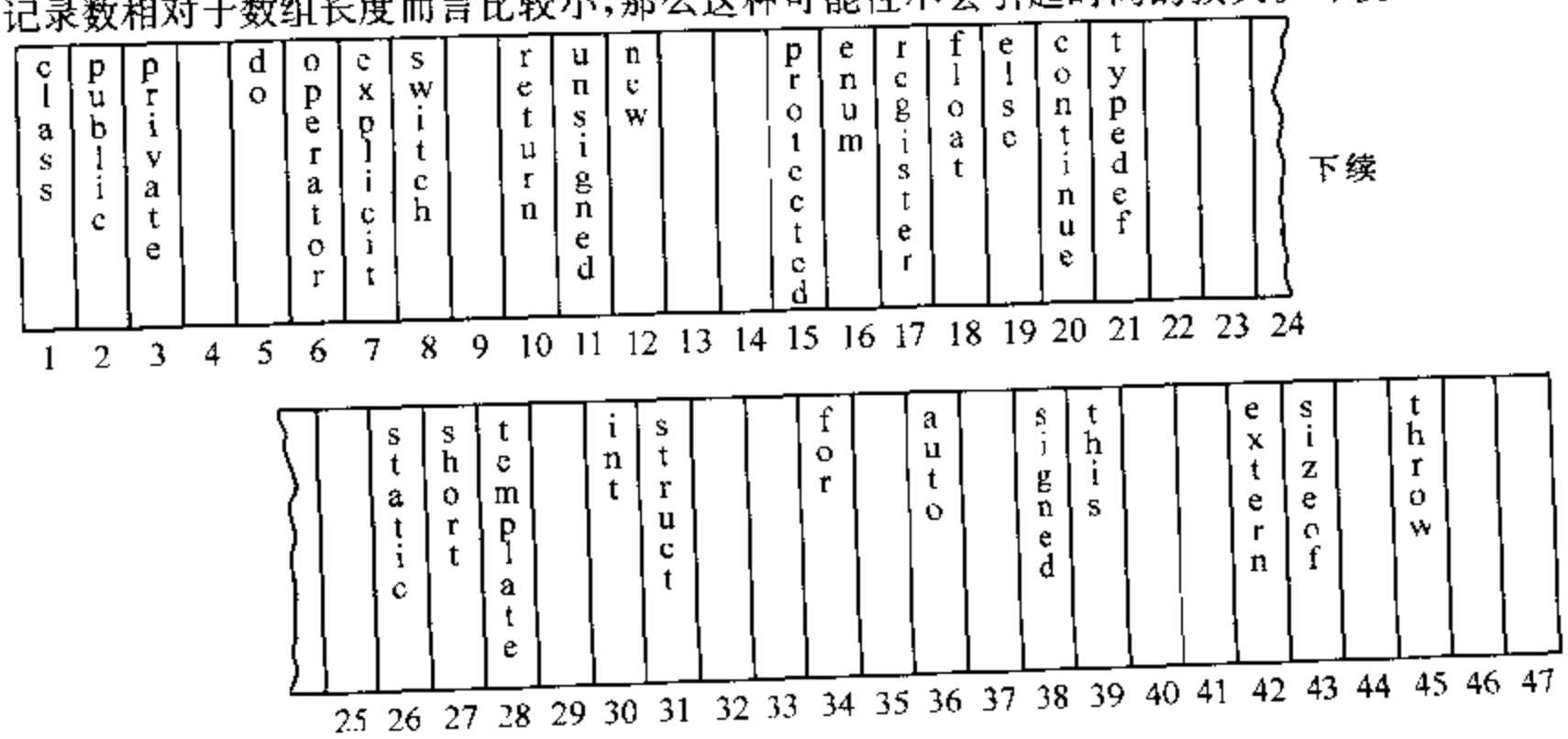


图 9.12　哈希表

的大多数元素被占据，哈希方法仍是信息检索的一种有效方法。

＊＊＊哈希函数

＊＊＊冲突

我们首先考虑哈希函数，它取一个键并将它映射到数组中的某个下标。这个函数通常会将几个不同的键映射到同一下标。如果期望的记录恰在下标给出的位置，那么我们的问题就得到解决，否则，必须使用某种方法解决两个记录定位于同一个位置而出现的冲突。因此在使用哈希法时必须回答两个问题：

- 首先，必须找到好的哈希函数；
- 其次，必须决定如何处理冲突。

在回答这些问题前，我们先暂停一下，非正式地列出实现哈希法所需要的步骤。

3. 算法概要

＊＊＊初始化

首先必须声明一个数组去容纳哈希表，接下来必须对数组中的所有位置初始化以表明它们为空，这个过程的完成依赖于应用，它的实现经常是在 Record 的成员中设置一个决不会作为真正的键出现的键。例如，对字母数字构成的键，由所有空格组成的键可能表达一个空位置。

＊＊＊插入

为了向哈希表中插入一个记录，首先要计算它的键的哈希函数。如果相应的位置是空的，则插入这个记录；如果键是相等的，则不允许插入新记录；其他情况（在这个位置已有一个具有不同键的记录）则有必要解决冲突。

＊＊＊检索

检索给定键的记录是完全类似的：首先计算这个键的哈希函数，如果期望的记录就在相应的位置中，则检索成功；当此位置不空且没有检查完所有的位置，则需要遵循用于冲突解决的同样步骤；如果发现空位或者已经检查完所有的位置，那么在表中就没有给定键的记录，这一次查找因此失败。

9.6.2 选择哈希函数

选择哈希函数的两个主要准则如下：

- 它应是简单的且易于快速计算；
- 它应实现在下标范围内真正出现的键的平均分布。

＊＊＊方法

如果我们事先就能准确地知道会出现什么键，则可能构造非常有效的哈希函数，但一般情况下我们事先并不知道会出现什么键。因此，通常构造哈希函数的方法是取得键，将它切开并以各种方式将切片混合在一起，从而得到（像计算机生成的伪随机数那样）下标范围内均匀分布的下标。

然而要注意，关于哈希函数不存在随机的成分，如果用同一个键去多次评价这个函数，那么它每次必然给出同样的结果，因此键的检索才不会失败。

哈希(hash)这个词正是来源于这个过程，因为这个过程生成的结果与原始的键完全没有相似性。同时，也希望打破出现在键中的任何模式和规律性，使得结果均一分布。即使哈希这个术语十分具有描述性，但在一些书中是用更技术性的术语如分散存储(scatter-storage)或键变换(key-transformation)来替代它。

我们考虑 3 种方法，它们能以各种方式放在一起构造哈希函数。

1. 截取

忽略键的一部分，并用剩余部分直接作为下标(认为非数字成员是其数字代码)。例如，如果这些键是 8 位整数，而哈希表有 1000 个位置，则从右边取第 1 位、第 2 位和第 5 位数字作为哈希函数，这样 21 296 876 就映射到 976。截取是一种非常快的方法，但它经常不能在表中均匀地分布这些键。

2. 折叠

将键划分成几个部分并用一种方便的方式(经常使用加法或乘法)将这几部分组合起来以得到下标。例如 8 位整数能被分成 3 组：3 位、3 位和 2 位，再将这 3 组一起加起来，如果必要的话还可以在适当的下标范围内截取这个和。因此 21 296 876 映射到 212+968+76=1256，截取后得到 256。由于键中所有的信息都能够影响函数的值，因此折叠法经常能比截取法获得更好的下标分布。

3. 模运算

*** 质数模

将键转换成一个整数(按需使用前述策略)，用下标范围的大小整除，并取余数作为结果，这等于使用C++的%运算符。取余数得到的分布很大程度上依赖于模(此情况下是哈希数组的大小)。如果模是一个小整数，如 2 或 10 的幂，则许多键趋于映射到同一个下标，而其他的下标却保留未用。模的最佳选择经常是一个质数，但也不总是质数，质数通常能很均匀地分布键(我们在后面将看到，质数模也改进了解决冲突的一种重要方法)。因此，在选择哈希表的长度时，通常不是选择 1000，较好的方法是选择 997 或 1009，$2^{10}=1024$ 通常是一种低劣的选择。取余数通常是推断计算哈希函数的最佳方式，因为它能获得较好的分布，同时也保证结果是在适当的范围内。

4. C++例子

*** 字母数字串

作为一个简单的例子，我们用C++编写一个哈希函数，将由 8 个字母数字构成的键转换成 0 到 hash_size-1 范围内的一个整数。因此，我们假设有 Key 这个类，它具有如下定义的方法和函数：

```
class Key:public String{
public:
  char key_letter(int position)const;
  void make_blank();
  //  Add constructors and other methods.
```

```
};
```

为了在实现这个类时节省一些程序设计工作，我们已选择继承第 6 章的 String 类的方法。这样做特别节省了对比较运算的编程。方法 key_letter(int position)必须返回 Key 的特定位置 position 中的字符，或者当 Key 的长度小于 n 时返回一个空格。最后的方法 make_blank 建立一个空的 Key。

我们现在能编写如下的简单的哈希函数：

```
* * * 简单的哈希函数

int hash(const Key &target)
/*Post: target has been hashed, returning a value between 0 and hash_size-1.
  Uses: Methods for the classKey. */
{
  int value=0;
  for (int position=0;position<8;position ++)
    value= 4*value+target.key letter(position);
  return value% hash_size;
}
```

我们只是将对应于 8 个字符中每个字符的整数代码相加，每次都乘以 4。然而，没有理由相信这种方法会比许多其他的方法更好(或更差)。例如我们能够减去其中一些代码，按对将它们相乘，或每隔一个字符就忽略一个。有时候应用程序将暗示一种哈希函数比另一种更好，有时候需要实验去确定一个好的哈希函数。

9.6.3 开放定址的冲突解决方案

1. 线性探测

最简单的解决冲突的方法是从哈希地址(发生冲突的位置)开始，在表中顺序查找期望的键或空位置，由于这种方法以直线方式查找，因此它被称为线性探测。这个表格将被看作环型的，这样当到达最后一个位置时，可以在表格的第一个位置继续查找。

2. 聚集

```
* * * 聚集的例子
```

线性探测主要的缺点是当表格达到大约半满时，存在聚集(clustering)的倾向，即记录开始出现在相邻位置的长字符串中，字符串之间存在间隙。因此为找到一个空位，顺序查找需要越来越长的时间。考虑图 9.13 中的例子，图中被占据的位置用浅灰色显示。假设数组中有 n 个位置，并且哈希函数以相等的概率 $1/n$ 选择其中任一位置。现在从一个相当均匀的分布开始，如顶图中所示。如果新的插入计算哈希后对应位置 b，则它将插入那里，但如果计算哈希后位于 a(a 已满)，则它也会插入 b。因此 b 被填满的概率是 $2/n$。下一阶段，向位置 a，b，c 或 d 的任一个插入的尝试会在 d 终止，因此 d 填满的概率是 $4/n$。此后，e 被填满的概率是 $5/n$，因此当发生另外的插入时，最可能的结果是使从位置 a 开始的满位置的字符串越来越长，因此哈希表的性能开始朝顺序查找的性能下降。

```
* * * 不稳定性
```

聚集问题本质上是一种不稳定性，如果一些键随机地碰巧彼此靠近，则其他的键越来越

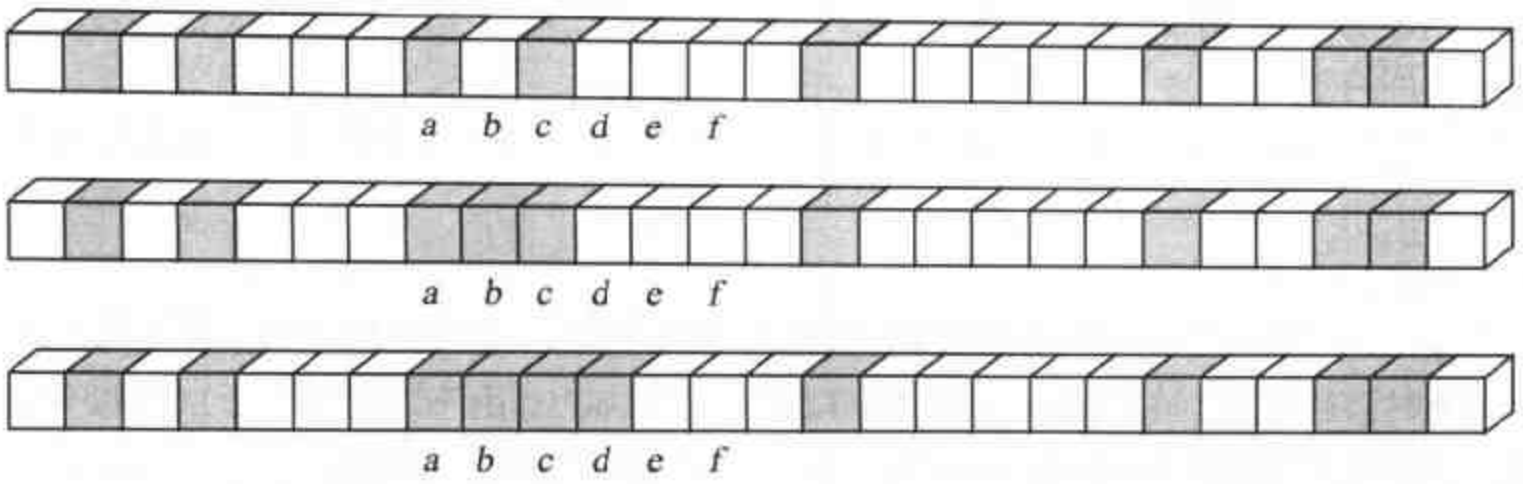


图 9.13　哈希表中的聚集

可能加入它们，因此导致分布逐渐变得更加失衡。

3. 增量函数

* * * 再哈希

如果我们想避免聚集这个问题，那么必须使用某种更复杂的方法去选择位置序列，从而检查何时会发生冲突。有很多检查冲突的方法，其中一种称为再哈希，它使用另一个哈希函数去获得供考虑的第二个位置。如果这个位置被填满，则需要某种其他方法去得到第三个位置，以此类推。但是如果从第一个哈希函数就能得到相当好的分布，则用独立的另一个哈希函数就获益不多。我们将同样找出一种更复杂的方法去确定从第一个哈希函数移动的距离，且无论第一个哈希函数的位置是什么都应用这种方法。因此我们希望设计一个增量函数，它能依赖于键或已经完成的探测次数，以此避免聚集。

4. 二次探测

如果在哈希地址 h 存在冲突，二次探测(quadratic probing)将探测表格中的位置 $h+1$，$h+4, h+9, \cdots$，即探测位置 $h+i^2$(换言之，此处的增量函数是 i^2)。

二次探测显著地降低了聚集，但却不能明显看出它是否会探测表格中的所有位置，实际上答案是否定的。对 hash_size 的某些值，这个函数会探测数组中比较少的位置。例如，当 hash_size 是 2 的一个较大的幂时，接近 1/6 的位置会被探测。然而，当 hash_size 是一个质数时，二次探测能达到数组中的半数位置。

* * * 证明

为证明这个观察结果，假设 hash_size 是一个质数，同时假设我们在探测 i 和某个随后的探测时到达同样的位置，对某个整数 $i>0$，取随后的探测为 $i+j$。假设 j 是最小的这样的整数，则此函数在 i 和在 $i+j$ 处计算的值相差 hash_size 的倍数。换言之，

$$h+i^2 \equiv h+(i+j)^2 (\text{bmod hash_size})$$

对这个表达式进行简化后，得到

$$j^2+2ij = j(j+2i) \equiv 0 \ (\text{mod hash_size})$$

后面这个表达式意味着 hash_size 整除(没有余数)j $(j+2i)$的积。质数能够整除一个积的惟一可能是整除它的一个因子，因此 hash_size 或者整除 j 或者整除 $j+2i$。如果出现第一种情况，则在再次探测 i 之前已做了 hash_size 次探测。(记住 j 是最小的正整数，使得探测 $i+j$ 是探测 i 的倍数。)当 $j=\text{hash_size}-2i$，或者当此表达式为负数时，第二种情况会在此表达式

增加了 hash size 的时候很快出现。因此将要探测的不同位置的总数正好是(hash_size+1)/2。

在这么多位置被探测后,习惯上可以认为这个表是满的,并且这个结果也相当令人满意。

* * * 计算

注意,二次探测可以不用乘法实现:在对位置 h 的第一次探测后,将增量设为 1,在每次相继探测时,在增量已被加到前一位置上以后将增量加 2,由于对所有 $i \geqslant 1$(可以使用数学归纳法证明这个事实),$1+3+5+\cdots+(2i-1)=i^2$,探测 i 将应要求查看位置 $h+1+3+\cdots+(2i-1)=h+i^2$。

5. 基于键的增量

我们能够让增量作为键自身的某个简单函数,而不是让它依赖于已做的探测次数。例如我们能够将键截成单个字符并用它的代码作为增量。在C++中,我们可以写成:

increment=(**int**)the_data. key_letter(0);

当将整除后的余数看作哈希函数时,一种较好的方法是让增量依赖于同一个除法的商。优化的编译器应只指定一次除法,因此这个计算会很快。在这种方法中,增量一旦确定就会保持常量。如果 hash_size 是一个质数,则随之可以得出:在重复之前,探测会经过数组中的所有元素,因此在数组完全满之前不会发生上溢。

6. 随机探测

最后一种方法是使用伪随机数生成器获得增量,所用的生成器应满足:只要用同样的种子(seed)总是生成同样的序列,①于是可以将这个种子指定为键的某个函数。这种方法能极好地避免聚集,但却可能比其他方法慢。

7. C++算法

为总结关于开放定址的讨论,我们继续学习已经介绍过的C++的例子(9.6.3),它使用字母数字键。假设 Key 和 Record 类具有我们上两节中用过的属性。特别地,我们假设这些类有 key_letter(int position)方法,它抽取键的特定 position 处的字符,并且有一个转换运算符提供 Record 的 Key。

我们用如下的声明建立哈希表:

```
const int hash_size=997;      //  a prime number of appropriate size
class Hash_table{
public:
  Hash_table();
  void clear();
  Error_code insert(const Record &new_entry);
  Error_code retrieve(const Key &target,Record &fcund) const;
private:
  Record table[hash_size];
```

① 见附录 B 中关于伪随机数生成器的讨论。

};

* * * 初始化

哈希表在创建时必须将数组 table 中的每个元素初始化为由 8 个空格组成的专用键.这是构造函数的任务,它的说明是:

```
Hash_table::Hash_table();
后置条件:创建哈希表并将它初始化为空。
```

还需要一个 clear 方法删除表中已存在的所有元素:

```
void Hash_table::clear();
后置条件:哈希表被清除并且为空。
```

虽然我们已开始说明哈希表的操作,我们不会继续去开发一个完整且通用的软件包。由于好的哈希函数的选择在很大程度上依赖于所用的键的种类,哈希表操作通常太依赖于特定的应用而不能被汇编成一个通用软件包。

为说明如何编写更多函数的代码,我们将继续接着 9.6.2 节编写的哈希函数的例子,并将二次探测用于冲突解决。我们已证明这样所做的探测的最大数目是(hash_size+1)/2,因此我们采用一个计数器 probe_count 去检查这个上界。

有了这些约定,我们现在来编写一个方法,它将一个 Record new_entry 插入到哈希表中。

* * * 插入

```
Error_code Hash_table::insert(const Record &new_entry)
/*Post: If the Hash_table is full, a code of overflow is returned. If the table already
      contains an item with the key of new_entry a code of duplicate_error is returned.
      Otherwise: The Record new_entry is inserted into the Hash_table and success is
      returned.
  Uses: Methods for classes Key, and Record. The function hash.*/
{
  Error_code result=success;
  int probe_count,          //  Counter to be sure that table is not full.
      increment,            //  Increment used for quadratic probing.
      probe;                //  Position currently probed in the hash table.
  Key null;                 //  Null key for comparison purposes.
  null.make_blank();
  probe=hash(new_entry);
  probe_count=0;
  increment=1;
  while (table[probe]!=null        //  Is the location empty?
        && table[probe]!=new_entry      //  Duplicate key?
        && probe_count<(hash_size+1)/2){    //  Has overflow occurred?
    probe_count++;
    probe=(probe+increment)% hash_size;
    increment+= 2;      //  Prepare increment for next iteration.
   }
   if (table[probe]==null)table[probe]=new_entry;  //  Insert new entry.
```

```
  else if (table[probe]==new_entry)result=duplicate_error;
  else result=overflow;     // The table is full.
  return result;
}
```

检索给定键的记录(如果有的话)的方法在形式上与上类似,因此将留作练习。检索方法将返回与 Key target 关联的完整的 Record,它的说明如下:

```
Error_ code Hash _ table:: retrieve (const Key &target, Record &found)
const;
```

后置条件:如果哈希表中的元素有与 target 相等的键,则 found 取这个元素的值并返回 success,否则,返回 not_present。

8. 删除

到目前为止,我们还未提及从哈希表中删除元素。乍看起来,它似乎是一项简单的任务,仅需用表示空键的专用键去标记被删除的位置。但这种方法却不起作用,原因在于空位被用作停止查找目标键的信号。假设删除前,存在一或两个冲突,并且某个元素的哈希地址等于现在已删除的位置,那么这个元素实际上存储在表格中的其他地方,如果我们试图检索那个元素,则现在的空位会停止这次查找,因此即使这个元素仍在表格中,还是不可能找到它。

*** 专用的键

纠正这个困难的一种方法是创造另一个专用的键,用于放到任何被删除的位置中。这个专用的键将指示这个位置可自由接受插入,但不用于终止表中某个其他键的查找。然而,使用这个附加的专用键会使算法稍微复杂且略慢。在我们迄今为止学习的有关哈希表的这些方法中,删除确实难以处理,因此应尽可能避免。

9.6.4 链式冲突解决方案

*** 链式存储

到目前为止,我们都是隐含地假设对哈希表操作时只使用顺序存储。实际上,由于我们希望能够快速引用哈希表中的随机位置,而链式存储不适于随机访问,因此顺序存储对哈希表本身来说是自然的选择。然而,没有原因说明对记录本身为什么不用链式存储。我们可以将哈希表本身当作一个链表数组,图 9.14 中给出了一个例子。

*** 链接

习惯上称哈希表中的链表为链,并称这种冲突解决方案为链接。

1. 链接的优点

*** 节省空间

这种观点有几个优点。第一个也是最重要的优点是当记录本身很大时可以节省很多空间。因为当哈希表是顺序数组时,必须在编译时留出足够的空间以避免溢出。如果记录本

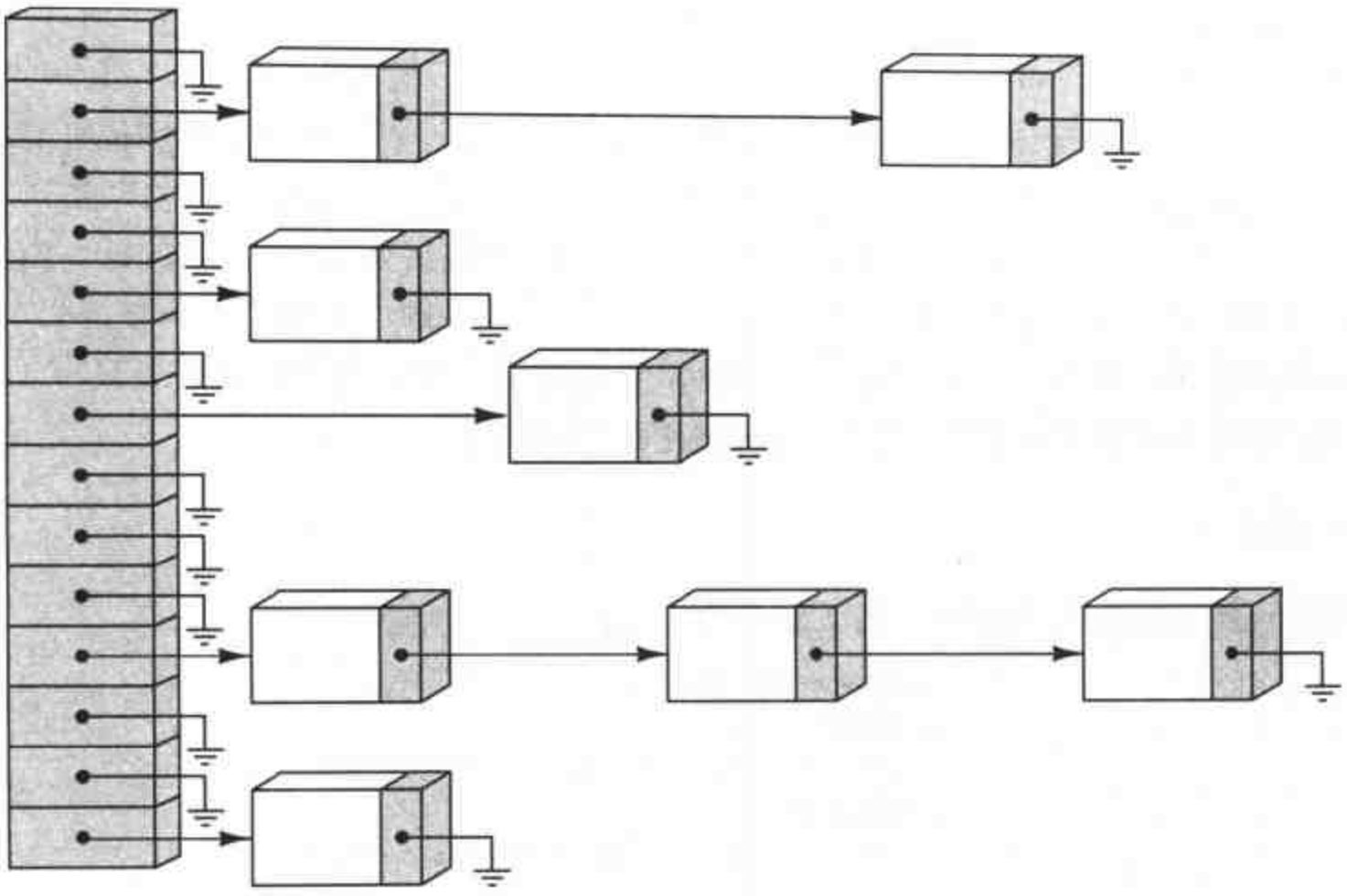


图 9.14　链式哈希表

身是在哈希表中，则如果有许多空位置（用于帮助消除冲突代价），它们会消耗很多空间，而这些空间却可能为其他地方所需要。另一方面，如果哈希表仅包含指向记录的指针，每个指针只需要一个字的空间，则哈希表的大小将减少一个很大的因子（基本上是等于记录大小的因子），因此它相对于记录或其他用途可用的空间是很小的。

＊＊＊冲突解决方案

在哈希表中仅保存链表的第二个主要优点是它允许简单有效的冲突处理。用一个较好的哈希函数，很少有键会给出同样的哈希地址，因此链表会很短且能快速查找。根本没有聚集问题，因为不同哈希地址的键总是转到不同的表。

＊＊＊溢出

第三个优点是不再需要哈希表的长度超出记录数。如果记录数比表格中的元素数还多，则仅意味某些链表确实包含不止一个记录。即使记录数是表格长度的几倍，链表的平均长度仍会保持较短，在适当的链表上进行顺序查找仍会保持较高的效率。

＊＊＊删除

最后，链式哈希表中的删除变成一项快速且简单的任务。从哈希表中删除的过程与从简单链表中删除完全一样。

2. 链接的缺点

＊＊＊空间的使用

链式哈希表的这些优点确实是强大的，但为了免于让你认为链接总是优于开放定址，我们也指出它的一个重大缺点：所有的链都需要空间。如果记录较大，则这点空间与记录本身所需的空间相比微不足道，但如果记录较小，则就不能忽略它。

* * *较小的记录

例如，假设每个链占用一个字，且每个元素本身仅占用一个字(只有键)。这样的应用相当普遍。如仅使用哈希表去回答一些关于键的"是/否"问题。假设我们使用链接并且哈希表本身很短，与元素数 n 一样具有 n 个元素。这样我们总共就要使用 $3n$ 个字的存储：n 个用于哈希表，n 个用于键，另外的 n 个用于每个链中查找下个结点(如果有的话)的链。由于哈希表接近满，这样将有许多冲突，且一些链上将有若干个元素，因而查找将有点慢。另一方面，假设我们使用开放定址，则同样 $3n$ 个字的存储全部用于哈希表意味着这个表只有 1/3 满，因此只有相对少的冲突，同时对给定元素的查找也会非常快。

3. C++算法

用C++描述的链式哈希表有如下简单的定义：

```
class Hash_table{
public:
  //  Specify methods here.
private:
  List<Record> table[hash size];
};
```

这里的 List 类可以是第 6 章学过的泛型链式表实现中的任一种。为求一致，链式哈希表的方法包括早先的哈希表实现中的所有方法。构造函数的实现仅是为数组中每个表调用构造函数。清除链式哈希表是一项完全不同的任务，为清除这个表格，我们必须清除表格每个位置上的链表，这个任务可用 List 的方法 clear()去完成。

我们甚至能使用表软件包中的方法去访问哈希表，哈希函数本身与开放定址中所用的并无区别。对数据检索来说，我们能够简单地使用 7.2 节中函数 sequential_search 的链式版本，Hash_table::retrieve 方法实质是：

```
sequential_search(table[hash(target)],target,position);
```

将它转换成完整函数的细节将留作练习。

类似地，插入的实质是：

```
table[hash(new_entry)].insert(0,new_entry);
```

这儿我们选择将新元素作为表的首结点插入，因为这是最简单的。正如所见，插入和检索都比开放定址的版本更为简单，因为冲突解决不再是一个问题，且我们能够利用前面对表所做的工作。

* * *删除

从链式哈希表中删除也比从开放定址哈希表中删除更为简单。为了删除给定键的元素，我们仅需要使用顺序查找找到它位于哈希表的相应链中的位置，然后将这个元素从它的链表中删除。这个函数的说明如下：

```
Error_code Hash_table::remove(const Key_type &target,Record&x);
后置条件：如果表格中有与 target 具有相等键的元素，则返回代码 success，元素从哈希
        表中删除并记录在 x 中，否则，返回代码 not_present。
```

相应函数的编写将留作一个练习。

练习 9.6

E1. 用数学归纳法证明：对所有的整数 $i>0$，$1+3+5+\cdots+(2i-1)=i^2$。

E2. 编写C++函数，将一个元素插入到使用线性探测的开放定址哈希表中。

E3. 编写C++函数，从使用(1)线性探测；(2)二次探测的开放定址哈希表中检索一个元素。

E4. 在一个学生项目设计中，所用的键为整数，这些键必须被转换成整数下标。其中一个学生认为他能够使用三角函数很好地混合这些键，因此他将哈希函数定义成(**int**) sin(n)。这种选择有什么错误？然后他决定用函数 exp(n)代替 sin(n)。评价这种选择。

E5. 设计一个简单的、易于计算的哈希函数，用于将 3 个字母的单词映射到 0 和 $n-1$ 之间的整数，其中包括 0 和 $n-1$。对 n=11，13，17，19，找出下面这些单词的函数值：

PAL LAP PAM MAP PAT PET SET SAT TAT BAT

尝试尽量减少冲突。

E6. 假设一个哈希表包含 hash_size=13 个元素，下标从 0 到 12，并且需要将下列键映射到表格中：10 100 32 45 58 126 3 29 200 400 0

(1)运用% hash_size 运算缩小这些键，确定它们的哈希地址并找出要发生多少冲突。

(2)将这些键的数字(按普通的十进制表示)相加并随后应用% hash_size，确定它们的哈希地址并找出要发生多少冲突。

(3)找出对这些键不发生冲突的一个哈希函数(固定键集合上的没有冲突的哈希函数是“理想的(perfect)”)。

(4)用 hash_size=11 重复这个练习的前述部分(固定键集合完全填满哈希表且没有冲突的哈希函数是“最小理想的(minimal perfect)”)。

E7. 用开放定址解决冲突的另一种方法是维护一个称为溢出表(overflow table)的独立的数组，这个表用于保存所有与占据位置发生冲突的元素。可以用另一个哈希函数插入它们，或者也可以用检索所用的顺序查找依次插入它们。讨论这种方法的优缺点。

E8. 编写用于处理链式哈希表的下列函数，使用 7.2 节的 sequential_search()函数和 6.1 节的表处理操作实现这些操作。

(1)Hash_table::Hash_table()

(2)Hash_table::clear()

(3)Hash_table::insert(const Record &new_entry)

(4)Hash_table::retrieve(const Key &target, Record &found)const;

(5)Hash_table::remove(const Key_type &target, Record &x)

E9. 为使用线性探测的开放定址哈希表编写一个删除算法，使用附加的专用键指示被删除的元素(见 9.6.3 节的第 8 部分)，并因此修改检索和插入算法。

E10. 采用线性探测，删除元素而不使用另一个特殊键是可能的。将被删除的元素标记为空。进行查找直到找到另一个空位。如果这次查找找到一个键，它的哈希地址是刚变空的位置或在该位置之前，则在那儿将它后移，使得它前一位置变空，并从

新的空位置继续这个过程。编写一个算法实现这个方法。检索和插入算法是否需要修改?

程序设计项目 9.6

P1. 考虑所有C++保留字的集合。[①] 将这些保留字看作是 16 个字符的串,少于 16 个字符长的保留字将在右边以空格填补。

(1)设计一个整数值的函数,当它应用于所有的保留字时会生成不同的值(你会发现编写一个从文件中读出这些字的短函数是有用的,应用你设计的函数,并确定出现什么冲突)。

(2)当你的函数产生的值用%hash_size 缩小时,找出最小的整数 hash_size,使得所有的值保持不同。

(3)(挑战)按需修改你的函数,直到前一部分中的 hash_size 达到与保留字的数目一样(于是你为C++保留字找到了一个最小理想的哈希函数,将这些字映射到表格中而没有留下空位)。

* * * 分子量

P2. 编写一个程序,它读入一个分子式,如 H_2SO_4,并输出它表达的化合物的分子量。你的程序应当能够处理如 $Al_2(SO_4)_3$ 中加括号的原子团(提示:使用递归找出加括号的原子团的分子量。简化:将整个分子式放在圆括号(...)中是有用的。你需要设立元素的原子量的哈希表,按它们的缩写进行索引。为简单起见,可以对这个表格进行限制,使其中的元素更通用。其中一些元素缩写为一个字母,而另一些缩写为两个字母。为取得一致,需要为一个字母的缩写加入空格)。

9.7 关于哈希的分析

1. 生日的惊奇

哈希冲突的可能性与下面这个著名的数学娱乐问题有关:在一个房间里需要有多少个随机挑选的人,才可能有两个人具有同样的生日(月和日)? 由于(除了闰年以外)有 365 个可能的生日,大多数人回答是几百人,但实际上答案却只有 23 人。

我们可以通过回答下面这个相反的问题去确定上面这个问题的概率:在一个房间里有 m 个随机挑选的人,没有两个人具有相同生日的概率是多大? 从任意一个人开始,在一本日历上划去这个人的生日。第二个人有不同生日的概率是 364/365,同样将这个人的生日划去。第三个人有不同生日的概率是 363/365。以此继续,我们发现如果前面的 $m-1$ 个人有不同的生日,则第 m 个人具有不同生日的概率是 $(365-m+1)/365$,由于不同人的生日之间是无关的,因此将这些概率相乘,我们得到 m 个人具有不同生日的概率是:

$$\frac{364}{365}\times\frac{363}{365}\times\frac{362}{365}\times\cdots\times\frac{365-m+1}{365}$$

① 关于C++的任何课本都将包含保留字的列表,但C++的不同版本支持不同的保留字集合。

＊＊＊概率

只要 $m \geqslant 23$，这个表达式就小于 0.5。

＊＊＊冲突很可能发生

关于冲突，生日惊奇问题告诉我们，凡是合理规模的任何问题，几乎肯定有某些冲突。因此我们的方法不仅要将冲突的数目最小化，而且在发生冲突时也要尽可能迅速地处理它们。

2. 计算探测次数

作为信息检索的另一种方法，我们想知道在定位给定目标键的过程中，对成功和不成功的查找平均分别进行多少次键比较。我们将使用“探测(probe)”这个词去查找一个元素并将它的键同目标键进行比较。

＊＊＊装填因子

需要进行探测的次数显然依赖于表格填满的程度，因此(如同查找方法)，我们设 n 表示表格中的元素数，t(它与 hash_size 是一样的)是容纳此表格的数组中的位置数，这个表格的装填因子(load factor)是 $\lambda = n/t$。因此 $\lambda = 0$ 意味着一个空表格；$\lambda = 0.5$ 意味这个表格是半满的。对于开放定址，λ 不能超过 1，但对于链接，λ 就没有什么限制。我们分别考虑链接和开放定址。

3. 关于链接的分析

对链式哈希表，我们在执行任何探测之前直接转到其中一个链表，假设包含目标(如果存在的话)的链有 k 个元素，注意 k 可以是 0。

＊＊＊不成功的检索

如果查找不成功，则目标需要与所有 k 个相应的键进行比较。由于这些元素均匀地分布在所有 t 个表上(出现于任何表上的概率相等)，被查找的链期望的元素数是 $\lambda = n/t$，因此不成功查找的平均探测次数是 λ。

＊＊＊成功的检索

现在假设查找是成功的。从关于顺序查找的分析，我们知道平均比较次数是 $\frac{1}{2}(k+1)$，这里 k 是包含目标的链的长度，但这个链的期望长度不再是 λ，因为我们事先知道它必须包含至少一个结点(目标)，除目标以外的 $n-1$ 个结点均匀地分布在所有 t 个链上，因此含有目标的链的期望长度是 $1+(n-1)/t$。除了长度很小的表格，我们可以用 $n/t=\lambda$ 去代替 $(n-1)/t$，因此成功查找的平均探测次数非常接近于 $\frac{1}{2}(k+1) \approx \frac{1}{2}(1+\lambda+1) = 1+\frac{1}{2}\lambda$。

总而言之：

> 对装填因子为 λ 的链式哈希表的检索，成功时的平均探测次数近似为 $1+\frac{1}{2}\lambda$，不成功时的平均探测次数近似为 λ 次。

4. 关于开放定址的分析

* * * 随机探测

为分析开放定址中进行的探测次数，我们首先忽略聚集的问题，假设不仅第一次探测是随机的，而且在发生冲突后，在表格的所有其余位置进行的下一次探测也是随机的。实际上，我们假设表格足够大，使得所有的探测都能看作独立事件。

我们首先研究不成功的查找，第一次探测命中一个已占用单元的概率是 λ，即装填因子，探测命中空单元的概率是 $1-\lambda$，不成功的查找以恰好两次探测结束的概率，因此是 $\lambda(1-\lambda)$，而且类似地，在不成功的查找中，用恰好 k 次探测完成的概率是 $\lambda^{k-1}(1-\lambda)$。不成功查找中期望的探测次数 $U(\lambda)$ 因此是：

$$U(\lambda)=\sum_{k=1}^{\infty} k\lambda^{k-1}(1-\lambda)$$

* * * 不成功的检索

A.1 节中给出了求解这个公式的和的方法，我们因此得到：

$$U(\lambda)=\frac{1}{(1-\lambda)^2}(1-\lambda)=\frac{1}{1-\lambda}$$

* * * 成功的检索

为了计算成功查找所需的探测次数，我们指出所需的次数恰好比插入元素前进行不成功查找的探测次数多 1。现在我们认为表格开始时是空的，每次插入一个元素。随着这些元素的插入，装填因子慢慢地从 0 增加到其最终的值 λ。这种一步一步的增长近似处理成连续的增长，并且用积分代替求和是合理的。因此得出成功查找的平均探测次数近似于：

$$S(\lambda)=\frac{1}{\lambda}\int_0^{\lambda} U(\mu)\mathrm{d}\mu=\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$$

总而言之：

对随机探测开放定址和装填因子为 λ 的哈希表，成功的平均探测次数近似为 $\frac{1}{\lambda}\ln\frac{1}{1-\lambda}$，不成功的平均探测次数近似为 $\frac{1}{1-\lambda}$。

* * * 线性探测

对线性探测开放定址的哈希表可以进行类似的计算。在这种情况下，关于连续的探测是无关的假设不再合理。但其中的细节更加复杂，因此我们仅给出结果，完整的推导可查阅本章结尾处的参考书目。

从线性探测的开放定址和装填因子为 λ 的哈希表中检索，成功的平均探测次数近似为 $\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$，不成功的平均探测次数近似为 $\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$。

5. 理论比较

图 9.15 给出上述表达式在不同装填因子时的值。

装填因子	0.10	0.50	0.80	0.90	0.99	2.00
成功查找，期望的探测次数：						
链接	1.05	1.25	1.40	1.45	1.50	2.00
开放定址，随机探测	1.05	1.4	2.0	2.6	46	—
开放定址，线性探测	1.06	1.5	3.0	5.5	50.5	—
不成功查找，期望的探测次数：						
链接	0.10	0.50	0.80	0.90	0.99	2.00
开放定址，随机探测	1.1	2.0	5.0	10.0	100.	—
开放定址，线性探测	1.12	2.5	13.	50.	5000.	—

图 9.15　哈希方法的理论比较

从这个表格可以得出几个结论。首先，显然链接总是比开放定址需要更少的探测。另一方面，链表的遍历一般比数组访问更慢，特别是在键比较能快速完成的时候，这会降低链表的优势。当记录较大时，键比较需要大量的时间，这时链接就盛行起来。当不成功的查找比较普遍时，链接也有特别的优势，因为采用链接可以发现空表或短表，因此经常完全不需要进行任何键的比较即可表明查找是不成功的。

对开放定址哈希表的成功查找，比较简单的线性探测方法并不比更复杂的冲突解决方法明显地慢，至少在表格几乎填满之前会是这样。然而对不成功的查找，快速聚集导致线性探测性能下降到长顺序查找的水平。因此我们可以推断：如果查找更可能成功，并且装填因子适度，则线性探测非常令人满意，但在其他情况下则应使用另外的方法（如二次探测）。

6. 实验比较

记住图 9.15 中给出的计算只是近似值这一点很重要，也要记住的是，实践中没有什么东西是完全随机的，因此我们总能认为理论值和实际计算之间存在某些差别。因此为了进行比较，图 9.16 给出了使用 900 个键进行实验研究的结果，这些键都是 0 到 1 之间的伪随机数。

装填因子	0.1	0.5	0.8	0.9	0.99	2.0
成功查找，平均探测次数：						
链接	1.04	1.2	1.4	1.4	1.5	2.0
开放定址，二次探测	1.04	1.5	2.1	2.7	5.2	- -
开放定址，线性探测	1.05	1.6	3.4	6.2	21.3	- -
不成功查找，平均探测次数：						
链接	0.10	0.50	0.80	0.90	0.99	2.00
开放定址，二次探测	1.13	2.2	5.2	11.9	126.	- -
开放定址，线性探测	1.13	2.7	15.4	59.8	430.	-

图 9.16　哈希方法的实验结果

* * * 比较

如果比较图 9.15 和图 9.16 中的数，我们会发现链接的实验值基本上与理论值一样，二次探测的实验值与随机探测的理论结果非常接近，它们之间的差别可以简单地解释为二次

探测并非真正随机的事实。对线性探测来说，当表格相对空时，结果是类似的，但对近乎填满的表格，理论计算得到的近似结果与实验得到的有很大的差别，这说明了在数学中做出简单假设所带来的影响。

* * * 结论

与信息检索的其他方法相比，对所有这些数，需要指出的重要一点是它们只依赖于装填因子，而不依赖于表格中的绝对元素数。在 40 000 个可能位置上的 20 000 个元素的哈希表中检索，平均来说并不慢于在 40 个可能位置上的 20 个元素的表格的检索。采用顺序查找，1 000 倍长的表需要花 1 000 倍的时间去查找；用二分查找，这个比率降为 10（更精确地说，是 lg 1 000），但所需的时间仍然随长度增加，而在哈希表中则不是。

我们将检索 n 个元素的观察结果总结如下：

- 顺序查找是 $\Theta(n)$；
- 二分查找是 $\Theta(\log n)$；
- 哈希表检索是 $\Theta(1)$。

最后，我们强调设计好的哈希函数的重要性，哈希函数应快速执行且最广泛地分布键。如果哈希函数低劣，则哈希算法的性能会降到顺序查找的水平。

练习 9.7

E1. 假设哈希表中每个元素占用 s 个存储字（不包括使用链接所需的指针成员），其中指针所需的空间量为一个字。同时假设在哈希表中有 n 个占据的元素，而且哈希表有 t 个可能的位置（t 与 hash_size 一样），它包含了占据的位置和空的位置。

(1)如果使用开放定址，确定这个哈希表需要多少存储字。

(2)如果使用链接，则每个结点会需要 $s+1$ 个存储字，包括这个指针成员。这 n 个结点总共使用多少存储字？

(3)如果使用链接，则哈希表本身要使用多少个字？（回忆一下，用链接时，哈希表本身只包含指针，每个指针只需要一个字。）

(4)将前两个部分中的答案相加，以找出链接所需要的全部存储需求。

(5)如果 s 比较小（即元素有较小的长度），则开放定址对给定的装填因子 $\lambda=n/t$ 需要较少的内存；但是对较大的 s（较大的元素），链接方法总共要求较少的空间。找出 s 的盈亏平衡点，使这两种方法在此时使用同样多的合计存储量。你的答案应该依赖于装填因子 λ 的 s 的公式，但它不应直接包含数字 t 或 n。

(6)用从 0.05 到 0.95 范围内的 λ 值，评价你的公式并用图形表示其结果。

E2. 生日问题的答案令人惊奇的一个原因是它与明显相关的问题具有不同的答案。对下列问题，假设有 n 个人在房间里，且不考虑闰年。

(1)随机选择一个日期，房间里某个人的生日恰是此日期的概率是多大？

(2)房间里至少有两个人具有同样的随机生日的概率是多大？

(3)如果我们挑选一个人并找出他的生日，则房间中另一个人与这个人具有相同生日的概率是多大？

* * * 有序的哈希表

E3. 在链式哈希表中，假设谈论键的次序是有意义的，并且假设每个链中的结点按键的次序

存放。如果存在待查找的键，则一旦查找经过这个位置时能尽快地结束。在不成功查找的情况下，平均少做多少次探测？为在正确的位置上插入一个结点，平均需要多少次探测？将你的答案同课本中关于无序链接的情况导出的相应数字进行比较。

E4. 在我们关于链接的讨论中，哈希表本身只包含表，每个链是一个表。一种变化的方法是将每个链的第一个实际的元素放入哈希表自身当中(像开放定址一样，空位由一个不可能的键指示)。用给定的装填因子，计算这种方法对空间的影响，将这个方法作为每个元素中存储字数(不包括链)的函数。

程序设计项目 9.7

P1. 通过编写和运行测试程序去实现各种哈希表和装填因子，为你的计算机生成如图 9.16 的表格。

9.8　结论：方法的比较

在这一章和第 7 章，我们已考察了 4 种不同的信息检索方法：

- 顺序查找；
- 二分查找；
- 表格查找；
- 哈希方法。

*** 选择数据结构

如果询问其中哪一种方法最好，那么首先必须选择回答的标准，这些标准包括应用提出的需求和影响数据结构选择的其他因素，因为前两种方法只适用于表，后两种方法适用于表格。然而，在许多应用中，我们可以自由地为自己的数据结构选择表或表格。

*** 表格查找

关于速度和方便性，顺序表格中的普通查找肯定更优，但有许多应用程序无法应用这种方法，如表更可取或键稀疏的情况。当插入或删除比较频繁时，这种方法也不合适，因为在顺序存储中进行这些操作需要移动大量的信息。

另外 3 种方法中哪一种是最好的依赖于其他标准，比如数据的格式。

*** 其他方法

顺序查找肯定是我们介绍过的最灵活的方法，其数据可以任何次序存储，或者用顺序表示，或者用链式表示。二分查找的要求比较苛刻，它的键必须是有序的，且数据必须用随机访问表示(顺序存储)。哈希方法则要求更高，需要特有的键序以适于哈希表的信息检索，但这种键序对任何其他目的则一般没有什么用处。如果数据对人工检查是立即可用的，则某种次序是重要的，这样哈希表就不合适。

*** 不十分理想的结果

最后，关于不成功的查找存在一个问题。单独地看，顺序查找和哈希方法除了查找不成功外不给出更多的信息。二分查找能确定哪个数据有最接近目标的键，为此可能提供有用的信息。在第 10 章，我们将学习基于树的排序方法，它结合了二分查找的效率和链式结构

的灵活性。

9.9 应用:再访 Life 游戏

在第 1 章结束时,我们指出在 Conway 的 Life 游戏中对数组使用的边界有很强的限制性且不自然,Life 单元应该是在一个无边界的网格上。换言之,我们确实希望有这样的C++声明:

```
class Life{
public:
  //  methods
private:
  bool map[int][int];
  //  other data and auxiliary functions
};
```

* * * 无界数组

* * * 稀疏表格

当然这种声明是无效的。由于任何一个时候,在无界网格中实际只占用有限的单元数,我们实际上应将 Life 游戏的网格看成一个稀疏表格,这样哈希表证明是一种有吸引力的表示网格的方法。

9.9.1 算法的选择

在更精确地说明数据结构之前,我们考虑可能使用的基本算法。在最初的实现中,主函数不引用存储在计算机中的 Life 配置方式,因此我们完全不需修改这个函数。

```
int main()          //  Program to play Conway's game of Life.
/*Pre: The user supplies an initial configuration of living cells.
  Post: The program prints a sequence of pictures showing the changes in the
        configuration of living cells according to the rules for the game of Life.
  Uses: The class Life and its methods initialize(),print() and update().
        The functions instructions(),user_says_yes().*/
{
  Life configuration;
  instructions();
  configuration.initialize();
  configuration.print();
  cout<<"Continue viewing new generations?"<<endl;
  while (user_says_yes()){
    configuration.update();
    configuration.print();
    cout<<"Continue viewing new generations?"<<endl;
  }
}
```

我们需要重新编写 Life 的 update 方法,使它使用表格去查找单元的状态。对配置中任何给定的单元,我们能够通过查找每个邻居单元的状态去确定活的邻居的数目。因此,如果

我们选定即将到来的一代中可能活着的候选单元的一个小集合，则 update 方法能使用表格去准确地确定它们中的哪一个将变成活的。在任何 update 中，我们必须检查已经活的单元及它们的邻居。因而在我们的 update 实现中，我们遍历这些单元，通过使用表格确定它们的邻居数，且选择将在下一代活着的那些单元。

9.9.2　数据结构说明

我们已决定 Life 配置将包含一个哈希表用于查找单元的状态，然而我们需要遍历配置中活的单元。不难看出，遍历哈希表的元素通常效率很低，因此我们并入活的单元的一个 List 作为 Life 配置附加的数据成员。存储在 Life 配置的这个表和表格中的对象携带有关于各个单元的信息，我们将这些单元表示成称为 Cell 的结构的实例：每个 Cell 必须包含一对网格坐标，因此我们得到下面的定义：

```
struct Cell{
  Cell(){row=col=0;}      //  constructors
  Cell(int x,int y){row=x;col=y;}
  int row,col;            //  grid coordinates
};
```

在这个结构定义中，我们提供了构造函数的内联(inline)实现，因此不需要相应的代码文件。在C++中，我们可以将内联方法实现放入头文件中，而不妨害程序的独立编译。

* * * Cell 的创建

作为 Life 配置的一种扩展，第一次就将遇到位于其边缘上的单元。无论何时需要一个新的 Cell，都必须动态地创建它。因为 Cell 对象是动态创建的，所以仅能使用指针访问它们。

* * * 间接链表

此外，为了能够在 Cell 对象生命期结束时处置它，我们必须保留相应指针的一个记录。因此，我们实现 Life 配置的表成员存储指向单元的指针，这一结果在图 9.17 中说明。

表的每个结点因此包含两个指针：一个指针指向一个单元，而另一个指针指向表的下个结点。

* * * 查找 Cell 的坐标

要注意的是，给出一个指向单元的指针，就能确定相应的单元坐标：只要简单地跟随指向 Cell 对象的指针及它的 row 和 col 数据成员。因此我们能方便地将指向单元的指针作为哈希表中的记录存储，而由这些指针所决定的单元坐标则作为相应的键。

* * * 空间的使用

现在我们考虑如何在新的 Life 程序中使用哈希表，对它的实现我们能做出明智的选择。我们必须决定是采用开放定址还是链接。存储在表格中的元素只需要少量空间：每个元素只需要存储一个指向 Cell 的指针。由于表格的元素小，因此在我们的选择中勿需过多考虑空间因素。如果采用链接，为容纳必需的指针，每个记录的规模将增加 100%，而哈希表本身更小，因此能获得比开放定址更高的装填因子。如果采用开放定址，记录将更小，但需要在哈希表中留出更多的空间以避免长查找和可能的溢出。

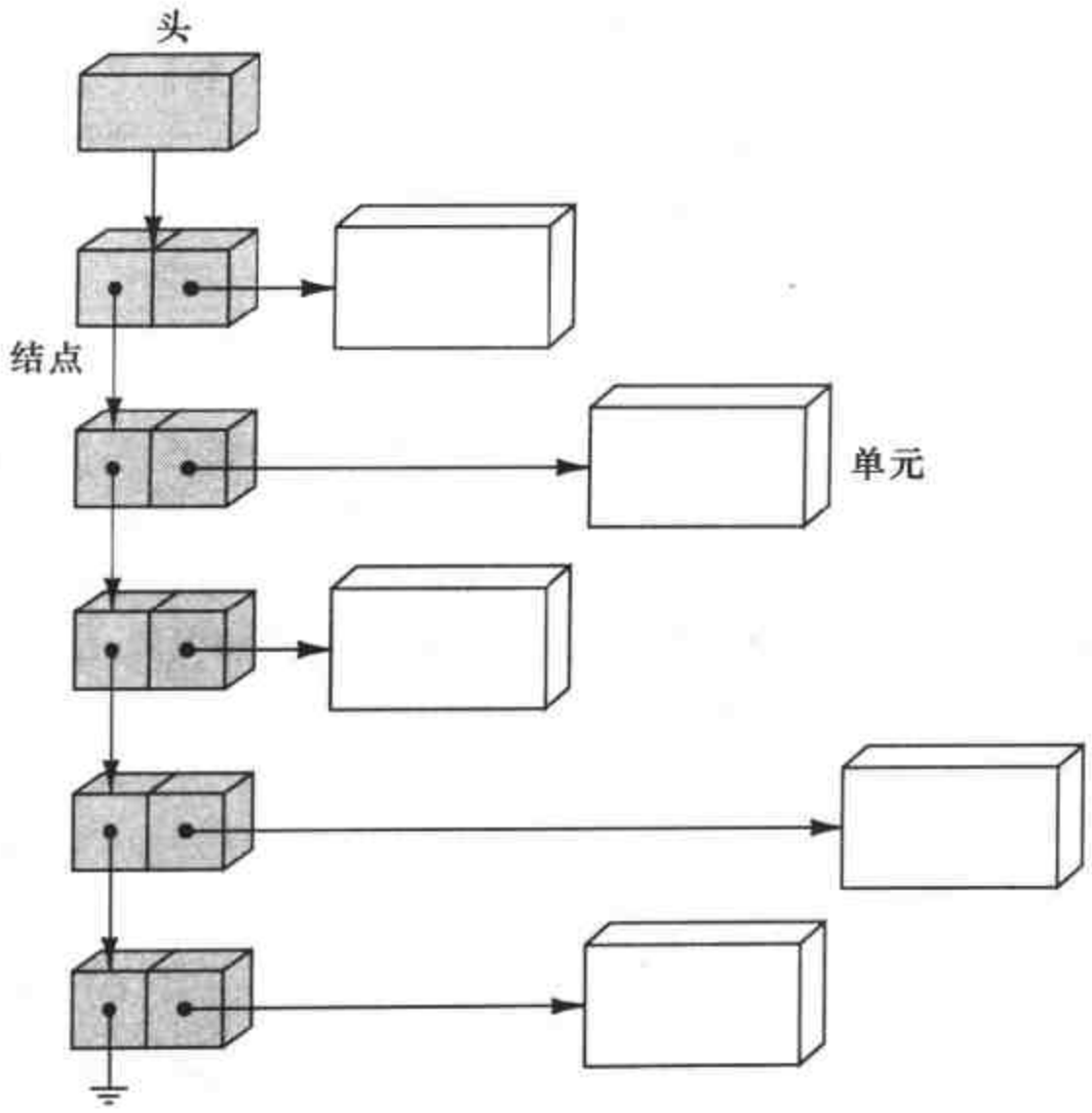


图 9.17 一个间接链表

* * *说明

因此，为了灵活性，我们决定使用链式哈希表，它具有如下的类定义：

```
class Hash_table{
public:
  Error_code insert(Cell*new_entry);
  bool retrieve(int row , int col) const;
private:
  List<Cell*> table[hash_size];
};
```

我们在这里说明了两个方法：insert 和 retrieve。我们做检索的惟一用途是去查询哈希表是否包含一个指向具有特定坐标的 Cell 的指针，因此，retrieve 方法使用一对坐标作为它的参数，并返回一个 bool 结果，指示表格中是否存在这样的一个 Cell。由于这个哈希表的方法的实现与 9.6 节讨论的那些方法非常类似，因此我们将它们留作一个设计项目。

请留意，Hash_table 确实带有默认的构造函数和析构方法，例如我们将依赖的析构函数对数组表格中的每个元素应用 List 的析构函数。

9.9.3 Life类

* * *指针操纵

做出这些决定后，我们现在就能限制 Life 类的表示和符号。为了便于用更新的版本去

替换配置，我们应间接地存储数据成员，例如指针。因此，Life 类需要构造函数和析构函数去分配和释放这些结构的动态存储。其他的 Life 方法——initialize，print 和 update——都被主函数 main 显式使用。

```
class Life{
public:
  Life();
  void initialize();
  void print();
  void update();
  ~Life();
private:
  List<Cell*> *living;
  Hash_table*is_living;
  bool retrieve(int row, int col)const;
  Error_code insert(int row, int col);
  int neighbor_count(int row, int col)const;
};
```

辅助成员函数 retrieve 和 neighbor_count 应用哈希表检索以确定单元的状态。另一个辅助函数 insert，创建一个动态的 Cell 对象，并同时将它插入到哈希表中和 Life 对象的单元表中。

9.9.4　Life 函数

为说明如何处理单元、表和表格，我们现在来编写 Life 的几个方法和函数，剩余的函数将留作练习。

1. 更新配置

＊＊＊复查第一个 Life 程序

至关重要的 Life 方法是 update，它的任务是从一种 Life 配置开始，并确定此配置在下一代将变成什么。在 1.4.4 节，我们是通过检查 grid 配置中的每个可能单元，计算它的邻居数以确定它在下一代是否是活的。此信息存储在局部变量 new_grid 中，它最后将被复制到 grid 中。

除了要使用一个无界的网格以外，我们继续延用这个算法框架。我们不必试着去检查配置中每个可能的单元，而是必须将自己的注意力限制在下一代中可能活着的单元。这些单元是哪些？当然，我们应检查所有活着的单元以确定它们中的哪个仍将活着，我们也应该检查一些死单元。如果一个死单元要变成活的，它必须恰好有 3 个活的邻居单元（按 1.2.1 节中的规则），因此如果我们检查活的单元的所有邻居单元时，要将所有满足上述条件的单元（和除此之外可能的其他单元）包含进来。所有这样的单元在图 9.18 的配置中以带阴影的缘饰显示。

在 update 方法中，逐步构造局部变量 Life new_configuration，用它表达即将产生的配置：我们循环检查当前配置中所有（活的）单元，同时也循环检查这些（活的）单元的所有（死的）邻居单元。对每个单元，首先必须确定它是否已被加入到 new_configuration 中，因为我

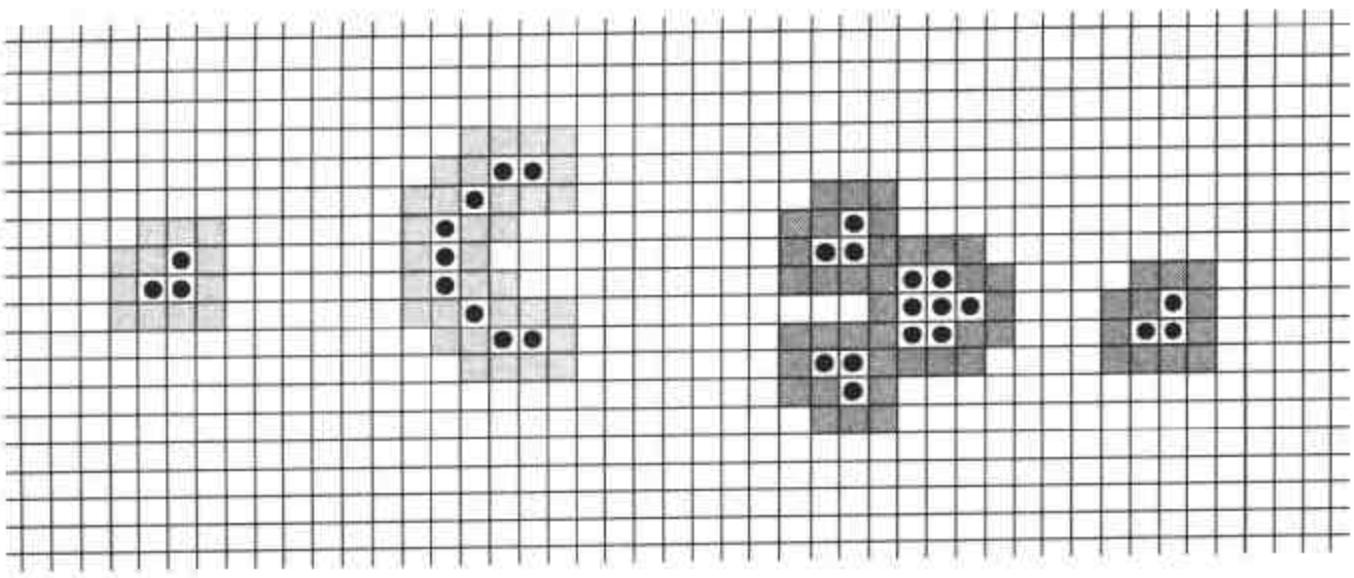


图 9.18 带死单元缘饰的 Life 配置

们必须足够小心，不能加入任何单元的完全相同的副本。如果此单元不属于 new_configuration，我们就使用 neighbor_count 函数确定是否应将它加入，如果适宜的话，就将它插入到 new_configuration 中。

在这个方法的最后，我们必须在当前配置和 new_configuration 之间交换 List 和 Hash_table 的成员，这个交换保证 Life 对象现在表示更新后的配置。此外，它保证了自动应用于局部变量 Life new_configuration 的析构函数将释放代表先前配置的单元、List 和 Hash_table。

我们因此得到下面的实现：

```
void Life::update()
/*Post: The Life object contains the next generation of configuration.
  Uses: The class Hash_table and the class Life and its auxiliary functions.*/
{
  Life new_configuration;
  Cell*old_cell;
  for (int i=0;i<living->size();i++){
    living->retrieve(i, old_cell);   //  Obtain a living cell.
    for (int row_add=-1;row_add<-2;row_add++)
      for (int col_add=-1;col_add<-2;col_add++){
        int new_row= old cell->row+row_add,
          new_col= old_cell->col+col_add;
//  new_row,new_col is now a living cell or a neighbor of a living cell,
        if (!new_configuration.retrieve(new_row, new_col))
          switch (neighbor_count(new_row,new_col)){
          case 3:          //  With neighbor count3, the cell becomes alive.
            new_configuration.insert(new_row, new_col);
            break;
          case 2:          //  With count2,cell keeps the same status.
            if (retrieve(new_row, new_col))
              new_configuration.insert(new_row,new_col);
            break;
          default:         //  Otherwise,the cell is dead.
            break;
          }
      }
  }
```

```
//  Exchange data of current configuration with data of new_configuration.
  List<Cell*> *temp_list=living;
  living=new_configuration.living;
  new_configuration.living=temp_list;
  Hash_table *temp_hash=is_living;
  is_living=new_configuration.is_living;
  new_configuration.is_living=temp_hash;
}
```

2. 输出

* * * 输出窗口

我们承认只能在用户屏幕上显示现在这个无界的 Life 配置的一小部分，因此我们仅输出一个矩形窗口，显示 Life 配置 20×80 中央部分的状态。对这个窗口中的每个单元，我们从哈希表中检索到它的状态，并因此输出空格或非空格字符。

```
void Life::print()
/*Post: A central window onto the Life object is displayed.
  Uses: The auxiliary function Life::retrieve.*/
{
  int row,col;
  cout<<endl<<"The current Life configuration is:"<<endl;
  for (row=0;row<20;row++){
    for (col=0;col<=80;col++)
      if (retrieve(row,col)) cout<<'*';
      else cout<<' ';
    cout<<endl;
  }
  cout<<endl;
}
```

3. 创建和插入新单元

* * * 任务

我们现在开始介绍 insert 函数，它创建一个 Cell 对象和对哈希表的显式引用。这个函数的任务是用给定的坐标创建一个新单元，并将它同时放入哈希表和表 List living 中。这个概要转换成下面的C++函数：

```
Error_code Life::insert(int row,int col)
/*Pre:  The cell with coordinates row and col does not belong to the Life configuration.
  Post: The cell has been added to the configuration.  It insertion into either the List
        or the Hash_table fails, an error code is returned.
  Uses: The class List, the class Hash_table, and the struct Cell*/
{
  Error_code outcome;
  Cell*new_cell=new Cell(row,col);
  int index=living->size();
  outcome=living->insert(index,new_cell);
  if (outcome==success)
    outcome=is living->insert(new_cell);
  if (outcone!=success)
```

```
    cout<<"Warning:new Cell insertion failed"<<endl;
  return outcome;
}
```

4. Life 对象的构造和析构

我们必须为自己的 Life 类提供构造函数和析构函数以分配和释放动态分配的成员。构造函数只需要应用 new 运算符。

```
Life::Life()
/*Post: The members of a Life object are dynamically allocated and initialized.
  Uses: The class Hash_table and the class List.*/
{
  living=new List<Cell*>;
  is_living=new Hash_table;
}
```

析构函数必须释放曾由 Life 类的方法动态定义的任何对象。除了 List * living 和 Hash_table * is_living 是由构造函数动态创建的以外，它们引用的 Cell 对象是由 insert 方法动态创建的，下面的实现由释放这些 Cell 对象开始：

```
Life::~Life()
/*Post: The dynamically allocated members of a Life object and all Cell objects that
        they reference are deleted.
  Uses: The class Hash_table and the class List.*/
{
  Cell*old_cell;
  for (int i=0;i<living->size();i++){
    living->retrieve(i,old_cell);
    delete old_cell;
  }
  delete is_living;          //  Calls the Hash table destructor
  delete living;             //  Calls the List destructor
}
```

5. 哈希函数

* * * 下标函数

我们的哈希函数与这一章早先的那些略微不同，原因在于它的参数已有两部分(行和列)，因此可以简单地做某种折叠。在决定如何实现前，我们先稍微考虑一下小型数组的特殊情况，其中的函数是一对一的且正好是下标函数。当每行正好有 maxrow 个元素时，下标 i,j 映射到 $i+\text{maxrow}*j$，将矩形数组一行接着一行地放到顺序的线性存储中。

在我们的哈希函数中使用类似的方案证明是有效的，这里我们用某个方便的数字(如一个质数)去代替 maxrow，它将把分布增加到最大限度并且减少冲突。因此我们得到

```
const int factor=101;
int hash(int row,int col)
/*Post: The function returns the hashed valued between 0 and hash_size-1 that corresponds
        to the given Cell parameter .*/
{
  int value;
```

```
  value=row+factor*col;
  value %=hash_size;
  if (value<0)return value+hash_size;
  else return value;
}
```

6. 其他子程序

剩余的 Life 成员函数 initialize,retrieve 和 neighbor_count 都与前述函数中的某一个或我们早先的 Life 程序的相应函数相当类似,因此这些函数确实能留作设计项目。

程序设计项目 9.9

P1. 编写 Life 方法 (1)neighbor_count;(2)retrieve;(3)initialize。

P2. 为存储指向 Life 配置单元的指针的链式实现编写 Hash_table 方法(1)insert;(2)retrieve。

P3. 修改 update 方法,让它使用另一个局部 Life 对象去存储已考虑插入但被拒绝的元素。使用这个对象去确保没有单元会被考虑两次。

启示和易犯的错误

1. 就像为算法所做的一样,对你的数据结构使用自顶向下的设计。首先确定数据的逻辑结构,然后慢慢地说明更多的细节,并尽可能延缓实现决策。
2. 在考虑详细的结构之前,决定数据上要求什么操作,且使用这个信息决定此数据是属于表还是表格。数据结构的遍历或者以预先指定的次序访问所有的数据通常隐含着选择表结构,而以时间 $O(1)$ 去访问任何元素通常隐含着选择表格。
3. 关于表的设计和编程,见第 6 章。
4. 使用数据的逻辑结构确定使用哪种表格:普通数组、某种特殊形态的表格、反转表格系统或哈希表。根据所要求的操作和空间的需求去选择最简单的数据结构。不要编写复杂的函数去节省不会被使用的空间。
5. 让数据的结构帮助你确定访问数据表格时是采用下标函数还是采用访问数组更好。只要可能,就使用嵌入在程序设计语言中的特征。
6. 在使用哈希表时,让数据的性质和必需的操作帮助你在链接和开放定址之间做出决策。链接在要求删除操作、记录相对较大或面临溢出问题时一般更可取;开放定址在单个记录较小及没有哈希表溢出的危险时一般更可取。
7. 哈希函数通常需要根据访问此哈希表的键的种类进行自定义。在设计哈希函数时,在保持哈希表上相对均匀的键分布的同时,尽可能保持简单且少量的计算。在计算中没有义务使用键的每个部分。对重要的应用,应当使用计算机实验哈希函数的几个变种,并寻找快速的计算和键的平均分布。
8. 回忆关于哈希的分析,某些冲突的发生是不可避免的,因此当键在表格中接近均匀分布时,不必担心冲突的存在。

9. 对开放定址来说，直到哈希表超过半满，聚集才会成为一个问题。如果表格比记录需要的空间大几倍，则线性探测将足够了，否则，需要更复杂的冲突解决方案。另一方面，如果表格比所需的大许多倍，则所有未用空间的初始化可能要求过多的时间。

复习题

9.1

1. 利用 Θ 和 Ω 符号，比较表格查找和表查找所需时间的区别。

9.2

2. 什么是行序为主序和列序为主序？

9.3

3. 锯齿表格为何需要访问数组而不是下标函数？
4. 反转表格的使用目的是什么？
5. 下标函数和访问数组之间如果有刻意的区别的话，这些区别是什么？

9.4

6. 什么操作对抽象表格是可用的？
7. 什么操作对表来说一般比表格更容易？

9.5

8. 用 20 个或更少的词，描述基数排序如何工作。
9. 在基数排序中，为何首先在最低有效位划分键，而不是在最高有效位？

9.6

10. 下标函数和哈希函数之间如果有刻意的区别的话，这些区别是什么？
11. 在设计哈希函数时应追求什么目标？
12. 列举 3 种经常嵌入在哈希函数中的技巧。
13. 什么是哈希表的聚集？
14. 描述两种最小化聚集的方法。
15. 列举链式哈希表相对于开放定址的 4 个优点。
16. 列举开放定址相对于哈希表的一个优点。

9.7

17. 如果哈希函数将 30 个键分配到长度为 300 的哈希表中的随机位置上，则有多大可能不会发生冲突？

进阶参考书目

本章的主要参考书目是 Knuth 的第 3 卷(见第 2 章的文献目录)，哈希是第 3 卷第 506～549 页的主题。Knuth 研究了我们提到的每种方法，另外还有许多其他的方法。他比我们更详细地进行算法分析，用伪汇编语言编写他的算法，并详细地计算操作次数。

下面的书(第 156～185 页)相当详细地讨论了各种数组、下标函数和访问数组：

C. C. Gotlieb and L. R. Gotlieb, *Data Types and Structures*, Prentice Hall, Englewood

Cliffs, N. J., 1978.

关于哈希函数和所用常量的选择的有趣的研究见：B. J. Mckenzie, R. Harries and T. C. Bell, "Selecting a hashing algorithm", *Software Practice and Experience* 20 (1990), 209 - 224.

下面这篇文章中讨论了生日惊奇问题的扩充：M. S. Klamkin and D. J. Newman, *Journal of Combinatorial Theory* 3 (1967), 279 - 282.

下面这本书的第 8 章中给出了关于哈希问题的全面且丰富的分析：Robert Sedgewick and Philippe Flajolet, *An Introduction to the Analysis of Algorithms*, Addison-Wesley, Reading, Mass., 1996.

第 10 章　二 叉 树

链表在数据结构的顺序表示上有很大的灵活性，但它们有一个弱点：它们是连续表，也就是说，它们的排列使得必须每次仅移动一个位置。这一章我们将克服这些不利之处，学习将树用作数据结构、使用指针和链表的方法去实现它们。将数据结构组织成树证明对一些应用是颇有价值的，特别是对信息检索问题。

10.1　二叉树

我们曾经用树说明算法的行为：在查找和排序算法中，我们画出比较树去说明键比较，画出子程序调用树，以及画出递归树。例如，如果考虑对下面的名字表运用二分查找，则比较的顺序可以用 10.1 中的比较树表示。

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

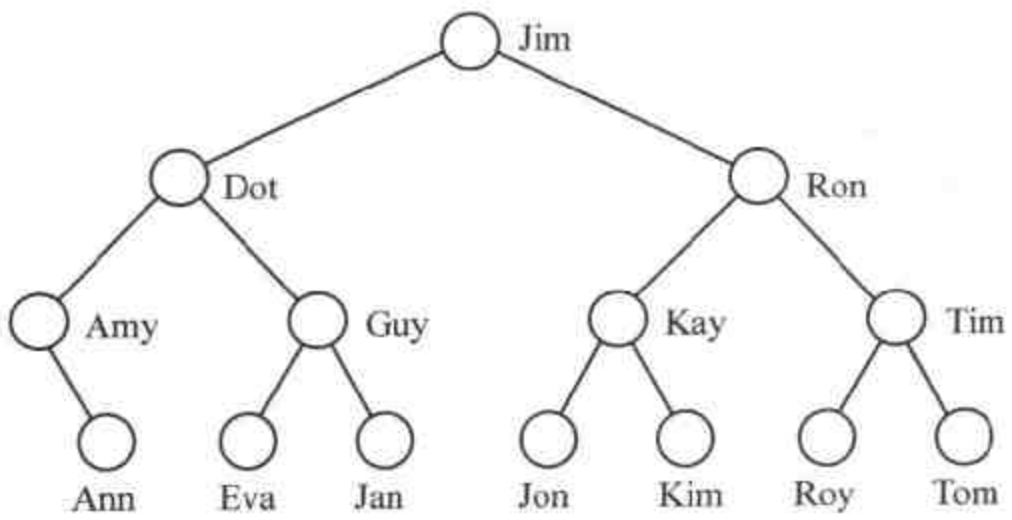


图 10.1　二分查找的比较树

10.1.1　定义

在二分查找中，当对键进行比较时，需要按照比较结果向左或向右移动，因此在我们建立的结构中保留左和右的关系显得很重要。也有可能某个给定结点的一侧或两侧的树部分为空。在图 10.1 的例子中，名字 Amy 的左子树为空，所有叶结点的左右子树都为空。

现在给出新的数据结构的形式化定义。

定义　一棵二叉树或者是空的，或者是由一个称为根(root)的结点和根的称为左子树(left subtree)和右子树(right subtree)的两棵二叉树组成。

＊＊＊ADT

注意这个定义是数学结构的定义。为了将二叉树说明成一种抽象数据类型，必须规定二叉树上能实施的操作。我们不急于给出这种定义，而是随着本章的进展设计这些操作。

还须注意的是，这个定义没有提及二叉树在内存中的实现方法。正如我们马上就可以看到的，采用链接表示显得自然而且容易使用，但其他的实现也是可能的。最后要注意的是，这个定义没有提及键及其排序的方法。除了查找，二叉树还用于多种目的。因此我们要保持定义的通用性。

在进一步考虑二叉树的通用特性前，我们回头去看一下通用定义，了解它的递归特征如何在小型二叉树的构造中产生效果。

＊＊＊小型二叉树

第一个例子不包含递归的基本情况，即空二叉树的情况。对其他种类的树，我们可能不会想到允许其为空，但对于二叉树来说，不仅在定义中而且在算法中，允许空树会给我们带来方便。空树通常是递归算法的基本情况，决定了算法什么时候终止。

构造只有一个结点的二叉树的惟一方法是使得那个结点成为树根，并且使它的左右子树都为空。因此不带分支的单个结点是且仅是一棵只有一个结点的二叉树。

当树中有两个结点时，其中的一个结点是根，另一个结点将在一棵子树中。因而左子树或右子树之一必须为空，而且另一棵将恰好包含一个结点。因此两个结点的二叉树有两种不同的形式。

＊＊＊左和右

在这一点上，要注意的是二叉树的概念与我们以前了解的一些树的例子有所区别，因为左和右对二叉树是重要的。两个结点的两棵二叉树可以画作：



它们互不相同。不能将二叉树的任何部分画得看起来像：



因为这样就没有办法知道低层的结点是其双亲的左孩子还是右孩子。

＊＊＊比较树

此外，应注意二叉树与我们在第 7 章和第 8 章算法分析中所学习的 2-树不属于同一类：2-树的每个结点或者有 0 个孩子或者有 2 个孩子，决不会是 1 个孩子，而这种情况在二叉树中却可以出现。左和右对于研究比较树的特性根本不重要，而在二叉树的研究中，它们

却是关键。①

* * *3 个结点的二叉树

对于 3 个结点的二叉树，其中一个结点为根，另外的结点将以下列方法中的一种划分左右子树：2+0 1+1 0+2。

由于第一种情况中左子树有两个结点，右子树是一棵空树，因此第一种情况给出了两棵二叉树。类似地，第二种情况也给出了两棵二叉树。第二种情况，左子树和右子树都各有一个结点，并且每棵二叉子树只有一个结点，因此在第二种情况下有一棵二叉树。于是 3 个结点的二叉树总共有 5 棵二叉树。

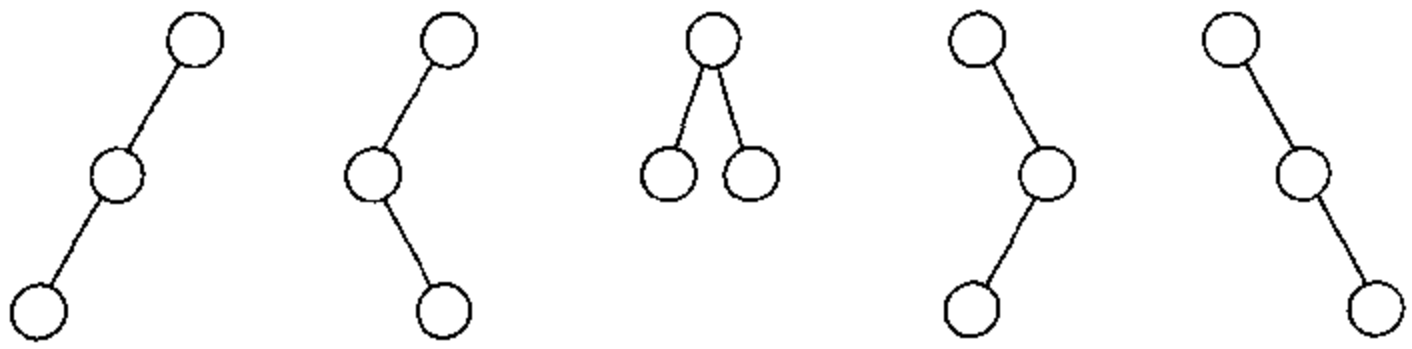


图 10.2 3 个结点的二叉树

图 10.2 显示了 3 个结点的二叉树。构造这些二叉树的步骤是用于构造结点数更多的二叉树的典型步骤。我们从根开始，考虑将剩余结点划分到左右子树中。左右子树则是较小的情况，我们从前面的工作中可以得到这些情况的结果。

在继续进行之前，我们暂停一下，先构造出 4 个结点的 14 棵二叉树。这个练习将进一步有助于建立二叉树定义后隐藏的思想。

10.1.2 二叉树的遍历

二叉树上最重要的操作是遍历，即周游二叉树中的所有结点，而且依次访问每个结点。至于其他数据结构的遍历，访问每个结点时要采用的动作将依赖于应用。

对表来说，它的结点从头到尾以自然的顺序出现，因此遍历以同样的顺序进行。然而对树来说，我们可以用许多不同的顺序去遍历所有的结点。当我们编写遍历二叉树的算法时，几乎总是希望继续下去以便在每个结点上都应用同样的规则，因此我们坚持采用一种通用的模式。

对一个给定的结点，我们希望以某种次序完成 3 个任务：访问结点本身、遍历它的左子树、遍历它的右子树。遍历次序间的关键区别是确定该在什么时候访问结点本身：是在遍历子树之前，在遍历子树之间还是在遍历所有子树之后。

如果将任务命名为：访问结点 V、遍历左子树 L 和遍历右子树 R，则得到六种排列方法：

V L R　L V R　L R V　V R L　R V L　R L V

① 然而在 10.3.6 节，我们将看到二叉树能转化为 2-树，反之亦然。

1. 标准的遍历次序

由标准的约定，仅考虑左子树在右子树前遍历的方式，则 6 种方法减为 3 种，这 3 种方法的镜像明显类似。从现在开始我们将使用左在右之前的 3 种方法，它们被赋予给定的特殊名字：

VLR	LVR	LRV
前序	中序	后序

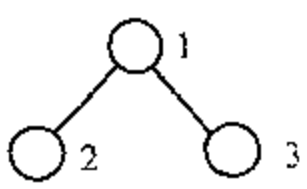
＊＊＊前序，中序和后序

这 3 个名字的选择是按照对给定结点访问的次序得到的。对于前序遍历(preorder traversal)，在访问子树之前访问结点；对于中序遍历(inorder traversal)，在访问左右子树之间访问结点；对于后序遍历(postorder traversal)，在访问两棵子树之后再访问根。

中序遍历有时称为对称次序(symmetric order)，且后序遍历曾经被称为末序(endorder)，我们将不使用这些术语。

2. 简单的例子

作为第一个例子，考虑下面的二叉树：



＊＊＊前序

在前序遍历中，根标号为 1，是第一个被遍历的结点。然后遍历就转到左子树，而左子树仅包含标号为 2 的结点，因此它被第二个遍历。然后前序遍历转到根的右子树，最后访问标号为 3 的结点。因此前序遍历访问的结点次序为 1,2,3。

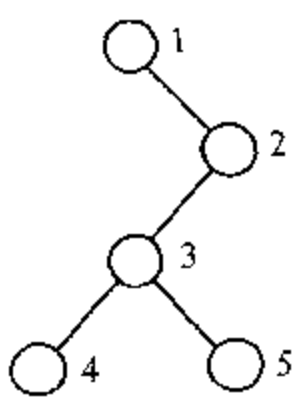
＊＊＊中序

在中序遍历中，访问根之前必须遍历它的左子树。因此，标号为 2 的结点第一个被遍历，这是根的左子树中惟一的一个结点，因此下一步，遍历将访问标号为 1 的根，最后是右子树。因此，中序遍历访问结点的次序为 2,1,3。

＊＊＊后序

在后序遍历中，在访问根之前先遍历左子树和右子树。首先访问左子树，它只有一个标号为 2 的结点，所以它是第一个被访问的结点。下一步，遍历右子树，访问结点 3，最后访问标号为 1 的根。因此，后序遍历访问结点的次序为 2,3,1。

作为第二个稍微复杂的例子，我们考虑如下的二叉树：



＊＊＊前序

首先，我们判断前序遍历。根的标号为1，是第一个访问的结点。下一步，遍历左子树，但这棵左子树为空，所以遍历的结果为空。最后，必须遍历根的右子树，这棵子树包含标号为2,3,4和5的结点。因此，我们又采用前序遍历方法去遍历这棵子树。因此接下来会访问这棵子树的根，它的标号为2，然后遍历2的左子树。在下一步，我们将遍历2的右子树，这棵右子树为空，所以什么也不做。但首先我们得访问左子树，它的根是3。对根为3的子树的前序遍历.访问的结点次序是3,4,5。最后，我们遍历2的为空的右子树。因此，对这棵树完整的前序遍历所访问的结点次序是1,2,3,4,5。

＊＊＊中序

对于中序遍历，必须从根的左子树开始，它为空。因此，标号为1的根是第一个访问的结点，然后遍历它的右子树，右子树的根是结点2。在访问结点2之前，必须遍历它的左子树，这棵左子树的根结点为3。根为3的这棵子树的中序遍历，访问的结点次序是4,3,5。最后，我们访问结点2并遍历它的右子树，这棵右子树为空，所以什么也不做。因此对这棵树完整的中序遍历的结点访问次序为:1,4,3,5,2。

＊＊＊后序

对后序遍历，必须在访问结点自身之前遍历其左右子树。因此，首先将遍历根为1的空的左子树，然后是右子树。二叉树的根总是后序遍历中最后访问的结点。在访问结点2之前，要遍历它的左右(为空)子树。根为3的子树后序遍历的次序为4,5,3。因此，对这棵树完整的后序遍历的结点访问次序为:4,5,3,2,1。

3. 表达式树

前序、中序和后序这3种最重要的遍历方法，其名字的选择不是偶然的，而是与表达式树这个相当重要的例子有着紧密的联系。

＊＊＊表达式树

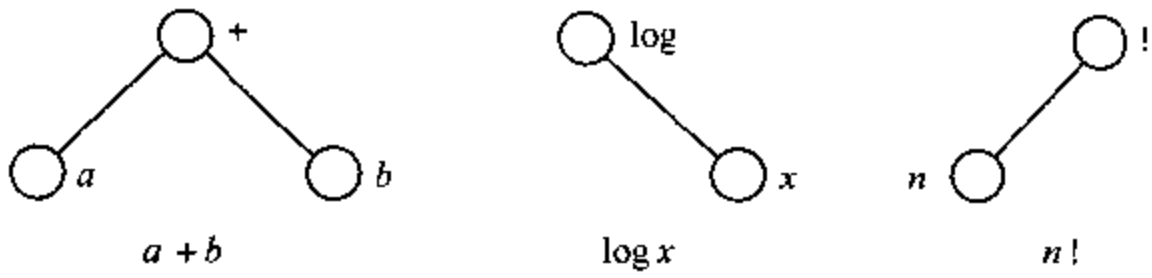
表达式树是由(算术或逻辑)表达式的简单操作数和运算符构建，通过将简单的操作数转换成二叉树的树叶，将运算符转换成内部结点而得到。对每一个二元运算符，左子树包含了给定运算符的左操作数中所有的简单操作数和运算符，右子树包含了右操作数中的每一项。

＊＊＊运算符

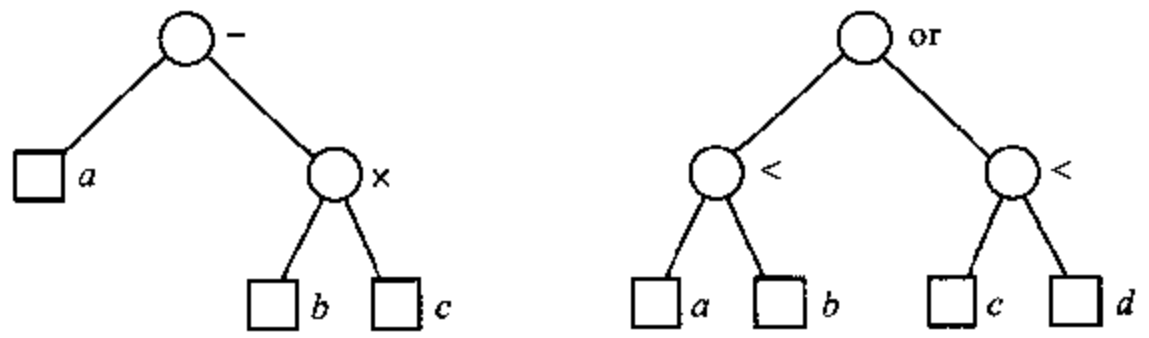
对一元运算符，两棵子树之一将是空的。按照惯例，我们将一些运算符写到它们的操作数的左边，如'?'(一元负)或标准函数像log()和cos()，其他的一元运算符写在操作数的右边，比如阶乘函数()! 或平方函数$()^2$。有时在两边中的任一边都是允许的，比如导数运算符，能够在左边写作d/dx，也能在右边写作$()'$，再如增量运算符++()。如果运算符写在左边，则在表达式树中其左子树为空，因此在树中操作数出现在运算符的右边，就像它们在表达式中一样。如果运算符出现在右边，则它的右子树将为空，且操作数将在运算符的左子树中。

图10.3是一些简单的表达式的表达式树，10.4是稍微复杂些的二次公式的例子，此处用^表示指数。

我们应该花一些时间用前序、中序和后序遍历这些表达式树中的每一棵。为了有助于

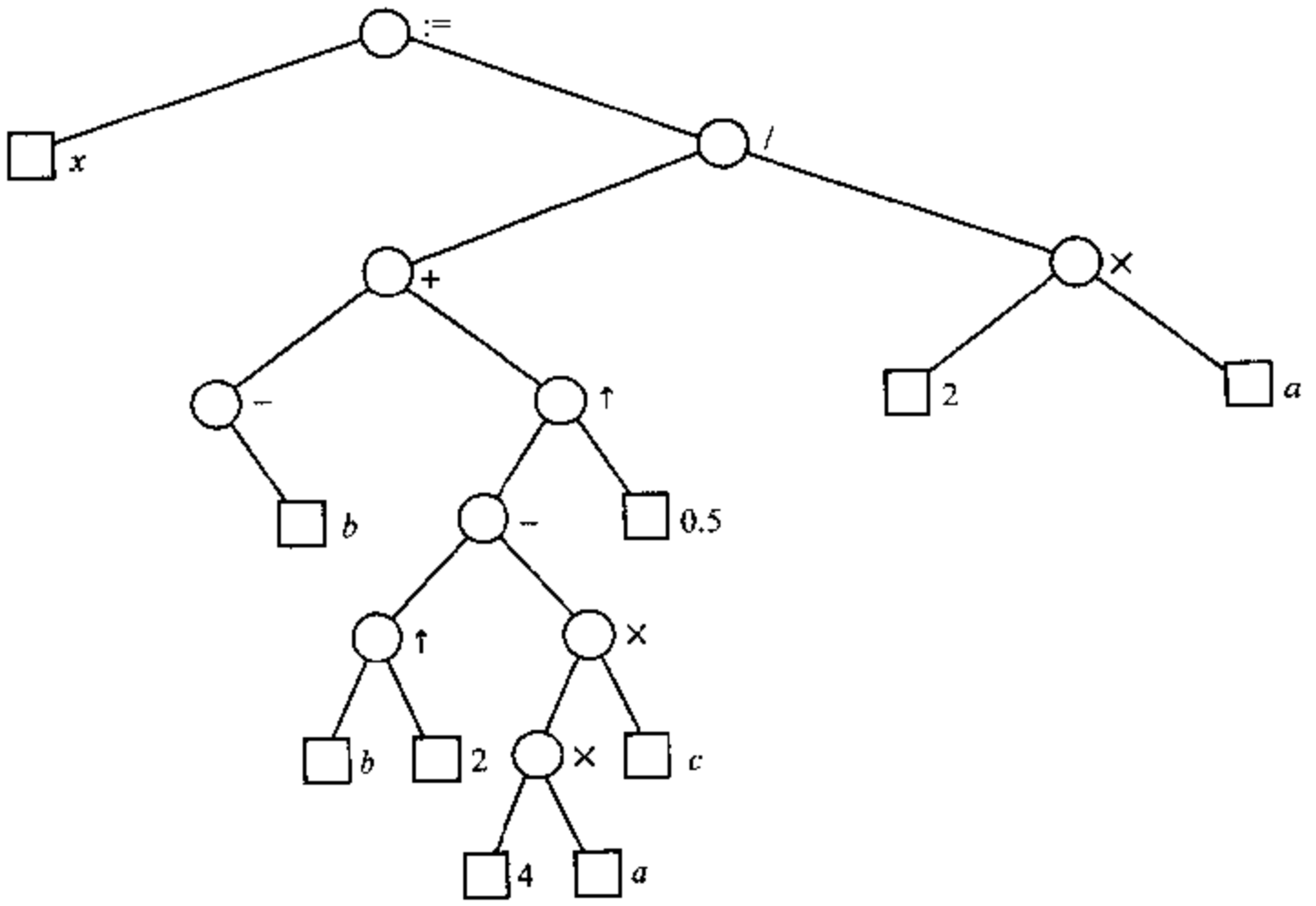


$a+b$　　$\log x$　　$n!$



$a-(b\times c)$　　$(a<b)\ \text{or}\ (c<d)$

图 10.3　表达式树



x:=(−b+(b↑2−4×a×c)↑0.5)/(2×a)

图 10.4　二次公式的表达式树

检查你所做的工作，图 10.5 中是这些遍历的结果。

表达式	$a+b$	$\log x$	$n!$	$a-(b\times c)$	$(a<b)\ \text{or}(c<d)$
前序	$+ab$	$\log x$	$!\ n$	$-a\times bc$	$\text{or}<ab<cd$
中序	$a+b$	$\log x$	$n!$	$a-b\times c$	$a<b\ \text{or}\ c<d$
后序	$ab+$	$x\ \log$	$n!$	$abc\times -$	$ab<cd<\ \text{or}$

图 10.5　表达式树的遍历次序

* * *波兰形式

遍历方法的名字与表达式的波兰形式有关：以前序遍历表达式树产生前缀式，其中每一个运算符写在操作数的前面；中序遍历给出了中缀式（书写表达式的习惯方法）；后缀遍历给出了后缀式，其中所有的运算符都出现在操作数后。考虑片刻后将使你深信这个原因：每个结点的左右子树是它的操作数，3 种波兰形式中运算符到其操作数的相对位置与用 3 种遍历方法中的每一种去访问这些成分的相对次序相同。波兰表示法是第 13 章的主题。

4. 比较树

作为进一步的例子，我们考虑图 10.1 中 14 个名字的二叉树（二分查找的比较树），并用 3 种遍历方法写出它们的次序：

前序：

Jim Dot Amy Ann Guy Eva Jan Ron Kay Jon Kim Tim Roy Tom

中序：

Amy Ann Dot Eva Guy Jan Jim Jon Kay Kim Ron Roy Tim Tom

后序：

Ann Amy Eva Jan Guy Dot Jon Kim Kay Roy Tom Tim Ron Jim

中序遍历产生按字母表顺序排列的名字并不是偶然的。图 10.1 中构造比较树的方法是：无论什么时候当目标键在所考虑的结点键之前，则移到左边，否则移到右边。因此二叉树的建立会使给定结点的左子树中的结点在序列中处于该结点之前，右子树中的所有结点在该结点之后出现。因此中序遍历首先产生了给定结点前的所有结点，然后是给定的结点，然后是后面的所有结点。

下一节将研究这种特性的二叉树，它们称为二叉查找树，因为它们对要求查找的问题很有用，而且具有较高的效率。

10.1.3 二叉树的链式实现

* * *根

在链式存储中，二叉树有自然的实现。就像对通常的链式结构那样，我们要将动态内存中创建的结点链到一起，因此需要一个独立的指针变量来找到这棵树。因为这个指针变量指向树的根，因此将它的名字定为 root。二叉树类的泛型模板的说明将采用如下形式：

```
template < class Entry>
class Binary_tree{
public:
//  Add methods here.
protected:
  //  Add auxiliary function prototypes here.
  Binary_node< Entry> *root;
};
```

像通常一样，模板参数 class Entry 将在客户代码里说明成实际类型。

我们现在考虑构成树的结点的表示。

1. 定义

二叉树的每个结点(作为某棵子树的根)有一棵左子树和一棵右子树,这些子树可以用指向它们的根结点的指针进行定位。因此我们得到下面的说明:

```
template <class Entry>
struct Binary_node{
//  data members:
  Entry data;
  Binary_node<Entry> *left;
  Binary_node<Entry> *right;
//  constructors:
  Binary_node();
  Binary_node(const Entry &x);
};
```

Binary_node 类包含两个构造函数,它们将任一新建的结点中的指针成员 left 和 right 设置为 NULL。

利用结点和树的说明,可以将这一节的第一个树图(图 10.1)中 14 个名字的比较树转换成图 10.6 的链式二叉树。不难看出,比较树和链式二叉树之间的惟一区别是在后者中明确地显示了 NULL 指针,而习惯上在画树的时候会省略所有空的子树及其附带的分支。

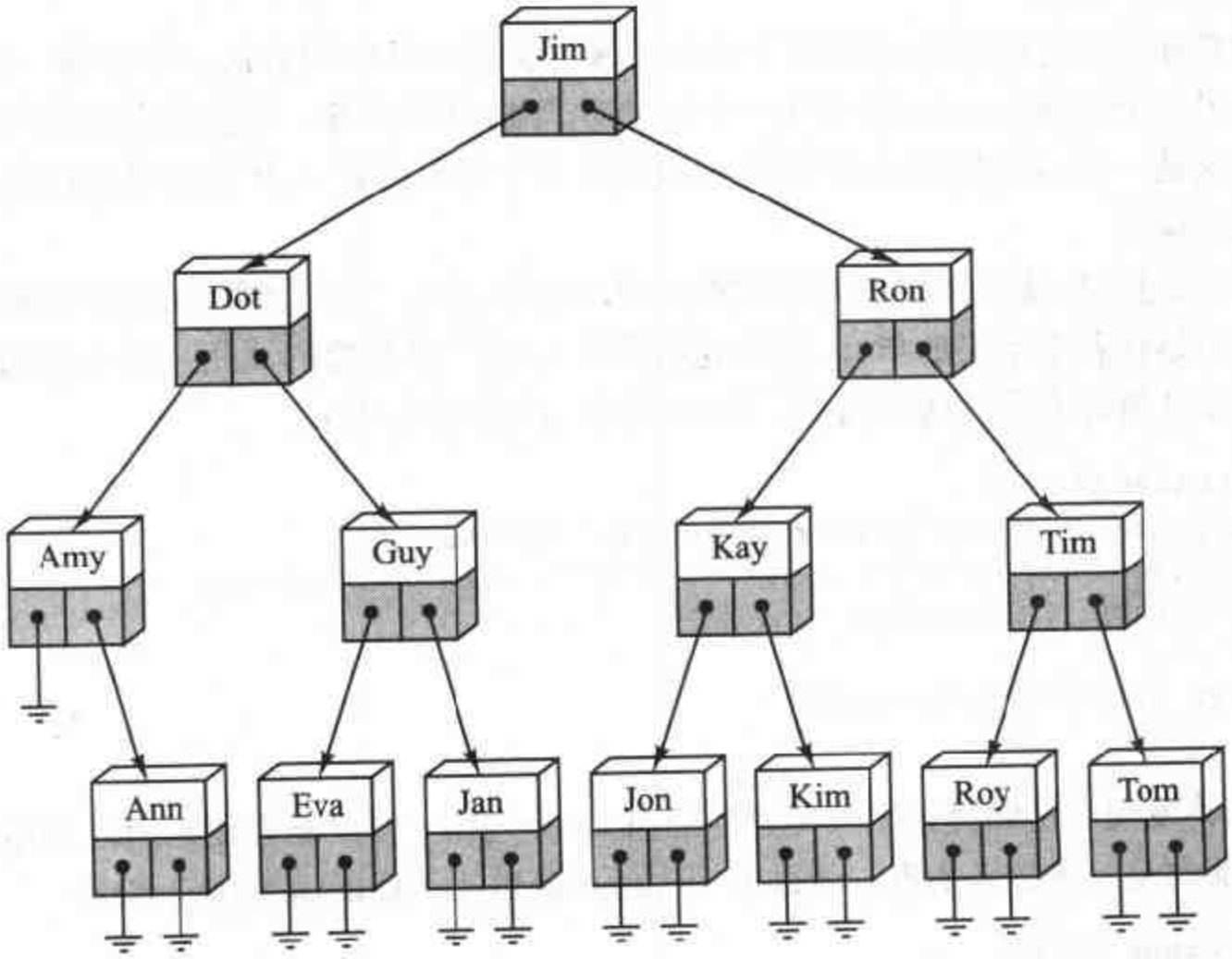


图 10.6　链式二叉树

2. 用于二叉树的基本方法

有了 root 指针,用下面的表达式就可以容易地识别空的二叉树:

root==NULL;

而且为了产生一个新的、空的二叉树，我们仅需要将 root 赋值为 NULL。Binary_tree 的构造函数只是形成这个赋值。

```
template <class Entry>
Binary_tree<Entry> ::Binary_tree()
/*Post: An empty binary tree has been created.*/
{
  root=NULL;
}
```

empty 方法测试 root 是否为 NULL，从而确定一棵 Binary_tree 是否为空。

```
template <class Entry>
bool Binary_tree<Entry>::empty()const
/*Post: A result of true is returned if the binary tree is empty. Otherwise, false is
        returned.*/
{
  return root==NULL;
}
```

3. 遍历

* * * 访问一个结点

我们现在来设计已经学过的用 3 种方式遍历链式二叉树的方法。如通常一样，我们将假设存在另一个函数 * visit，它对每一个结点执行期望的任务。就像为其他数据结构所定义的遍历函数一样，我们将为这个遍历函数安排一个函数指针 visit 作为形式参数。

* * * 递归遍历

在遍历函数里，我们需要访问根结点并遍历它的子树。递归将使得遍历子树极其简单。子树的定位是循着来自根的指针，因此这些指针必须传递给递归调用。每个遍历方法应调用一个带额外指针参数的递归函数。比如，中序遍历如下编写：

```
template <class Entry>
void Binary_tree<Entry>::inorder(void (*visit)(Entry &))
/*Post: The tree has been been traversed in infix order sequence.
  Uses: The function recursive_inorder*/
{
  recursive_inorder(root,visit);
}
```

通常，我们将发现由递归过程自然描述的任何 Binary_tree 方法能方便地通过调用一个应用于子树的辅助递归函数来实现。辅助的中序遍历函数用下面的简单递归实现：

```
template <class Entry>
void Binary_tree<Entry>::recursive_inorder (Binary_node<Entry> *sub_root,
                                            void (*visit) (Entry &))
/*Pre: sub_root is either NULL or points to a subtree of the Binary tree.
  Post: The subtree has been been traversed in inorder sequence.
  Uses: The function recursive_inorder recursively*/
{
  if (sub_root!=NULL){
```

```
    recursive_inorder(sub_root->left,visit);
    (*visit)(sub_root->data);
    recursive_inorder(sub_root->right,visit);
  }
}
```

其他的遍历方法类似地构造成对辅助递归函数的调用。辅助函数具有如下实现：

```
template <class Entry>
void Binary_tree<Entry>::recursive_preorder (Binary_node<Entry> *sub_root,
                                             void (*visit) (Entry &))
/*Pre: sub_root is either NULL or points to a subtree of the Binary_tree.
  Post: The subtree has been been traversed in preorder sequence.
  Uses: The function recursive_preorder recursively */
{
  if (sub_root!=NULL){
     (*visit)(sub_root->data);
     recursive_preorder(sub_root->left,visit);
     recursive_preorder(sub_root->right,visit);
  }
}

template <class Entry>
void Binary_tree<Entry>::recursive_postorder (Binary_node<Entry> *sub_root,
                                              void (*visit) (Entry &))
/*Pre: sub_root is either NULL or points to a subtree of the Binary_tree.
  Post: The subtree has been been traversed in postorder sequence.
  Uses: The function recursive_postorder recursively */
{
  if (sub_root!=NULL){
     recursive_postorder(sub_root->left,visit);
     recursive_postorder(sub_root->right,visit);
     (*visit)(sub_root->data);
  }
}
```

我们将标准的 Binary_tree 方法如 height,size 和 clear 的编码留作练习。其他方法通过调用递归辅助函数也最容易实现。在练习里，我们将设计将元素插入到 Binary_tree 中的方法。这个插入方法对测试我们基本的 Binary_tree 类是有用的。

我们将在这一章的后面创建一些更专门的、更有用的派生树类：这些派生类将有效地覆盖插入方法。派生类也将拥有删除元素的有效方法，但此时我们不把这些方法加入到基本的二叉树类中。这些决定导致 Binary_tree 类具有下面的说明：

```
template < class Entry>
class Binary_tree{
public:
  Binary_tree();
  bool empty() const;
  void preorder(void (*visit)(Entry &));
  void inorder(void (*visit)(Entry &));
  void postorder(void (*visit)(Entry &));
  int size()const;
  void clear();
  int height()const;
```

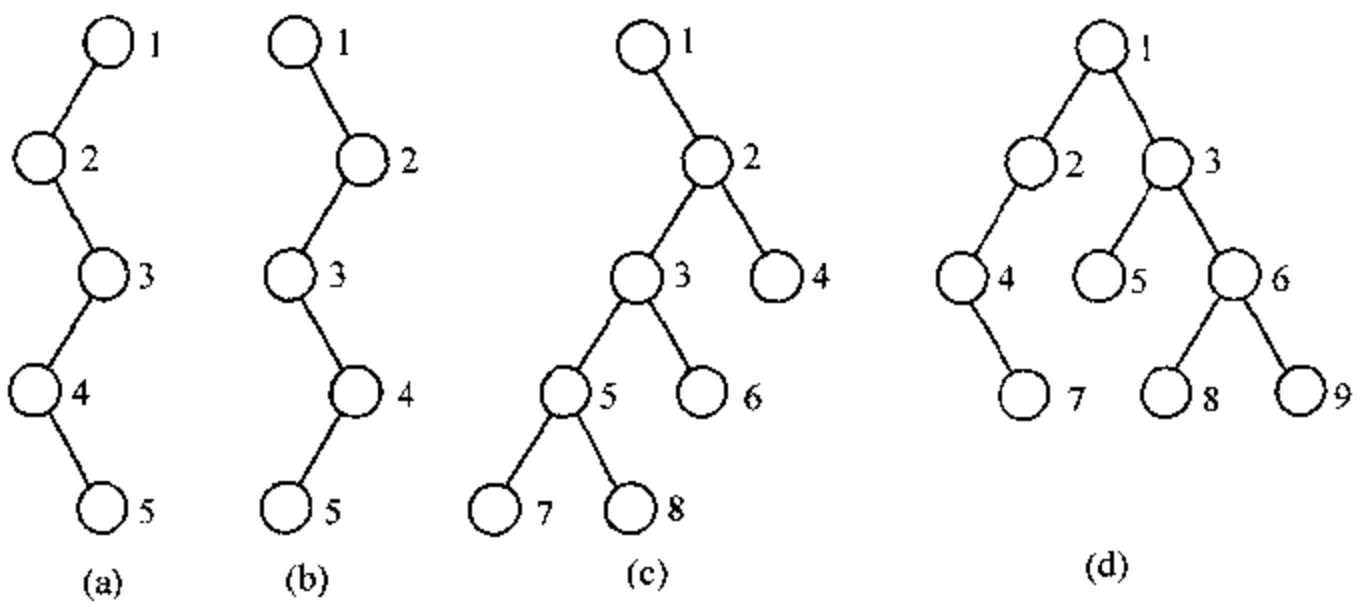
```
void insert(const Entry &);
  Binary_tree(const Binary_tree<Entry>&original);
  Binary_tree & operator=(const Binary_tree<Entry>&original);
  ~Binary_tree();
protected:
  //  Add auxiliary function prototypes here.
  Binary_node<Entry>*root;
};
```

虽然我们的 Binary_tree 类显现出来仅是一个壳，它的方法仅仅是将工作传递给辅助函数，但它却服务于一个重要的目的。这个类将各种树的方法集中在一起，提供了一个很方便的类似于其他 ADT 的客户接口。此外，这个类还提供了封装：没有它，树数据将得不到保护而容易被破坏。最后，我们会看到这个类将用作其他更有用的、派生二叉树类的基类。

练习 10.1

E1. 构造 4 个结点的 14 棵二叉树。

E2. 对下列二叉树的遍历，确定结点访问次序：(1)前序；(2)中序；(3)后序。



E3. 对下列表达式，画出表达式树，并说明访问的结点次序(1)前序；(2)中序；(3)后序：

(1)$\log n!$　　(3)$a-(b-c)$

(2)$(a-b)-c$　　(4)$(a<b)$ and $(b<c)$ and $(c<d)$

* * * Binary_tree 的大小

E4. 编写方法和相应的递归函数计算链式二叉树的所有结点数。

E5. 编写方法和相应的递归函数计算链式二叉树的树叶数(也就是，两棵子树都为空的结点)。

E6. 编写方法和相应的递归函数找出链式二叉树的高度，此处空树的高度为 0，仅有一个结点的树的高度为 1。

* * * Binary_tree 的插入

E7. 编写方法和相应的递归函数，将作为参数传递的 Entry 插入到链式二叉树中。如果 root 为空，则将新元素插入为根，否则将它插入到根的两棵子树中较短的一棵(或者如果两棵子树具有同样的高度，则将它插入到左子树中)。

* * * Binary_tree 的清除，Binary_tree 的析构函数

E8. 编写方法和相关的递归函数遍历二叉树(无论以什么方式只要方便就行)并且释放它所有的结点。使用这个方法实现 Binary_tree 的析构函数。

* * * Binary_tree 拷贝构造函数

E9. 编写生成链式二叉树副本的拷贝构造函数：

```
Binary_tree<Entry> :: Binary_tree(const Binary_tree<Entry>&original)
```

这个拷贝构造函数从系统获得必要的新结点且将原树的结点复制到新树。

* * * Binary_tree 赋值运算符

E10. 编写重载二叉树赋值运算符的方法：

```
Binary_tree<Entry>&Binary_tree<Entry> :: operator=
                        (const Binary_tree<Entry>&original)
```

* * * 双序遍历

E11. 编写函数执行二叉树的双序遍历，也即对树的每个结点，函数先访问这个结点，然后遍历它的左子树(以双序)，然后再次访问结点，然后遍历它的右子树(以双序)。

E12. 对练习 E2 中的每棵二叉树，通过调用方法 A 得到的混合顺序访问结点，给出结点的次序：

```
void Binary_tree<Entry> ::
  A(void (*visit)(Entry&))
{
  if (root!=NULL){
    (*visit)(root-> data);
    root-> left.B(visit);
    root-> right.B(visit);
  }
}
```

```
void Binary_tree<Entry> ::
  B(void (*visit)(Entry&))
{
  if (root!= NULL){
    root->left.A(visit);
    (*visit)(root-> data);
    root-> right.A(visit)
  }
}
```

* * * 打印二叉树

E13. (1)假设 Entry 的类型为 char。编写函数，它将以括号表示形式(data:LT,RT)打印二叉树中的所有元素，此处 data 是根的 Entry，LT 表示以括号表示形式打印的根的左子树，RT 表示以括号表示形式打印的根的右子树。比如，图 10.3 的第一棵树打印为：

(+:(a:(:,),(:,)),(b:(:,),(:,)))

(2)修改函数使得它不打印空树，而不是将空树打印成(:,)，对于仅仅包含一个元素 x 的结点，打印成 x，而不是(x:,)。因此先前的树现在打印为(+:a,b)。

E14. 编写函数交换链式二叉树的左右子树，见 10.7 中的例子。

* * * 层次遍历

E15. 编写按层次遍历二叉树的函数。也就是，首先访问根，然后是根直接的孩子，然后是根的孙子等(提示：使用队列追踪结点的孩子，直到访问到它们。图 10.7 的第一棵子树的结点以层次顺序编号)。

* * * 宽度

E16. 编写函数，它返回链接二叉树的宽度，也就是说，同一层中结点的最大数目。

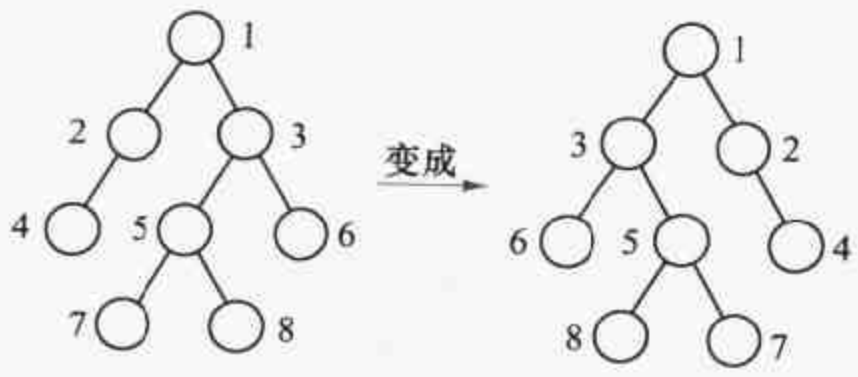


图 10.7 二叉树的反转

＊＊＊遍历序列

对以下的练习，假设存储在二叉树结点中的数据截然不同，但并不假设树是二叉查找树。也就是说，数据的任意次序和它们在树中的位置之间没有必然的关系。如果以特定的次序遍历一棵树，当访问到它的结点时就输出键，结果得到的序列称作是与那个遍历相符的序列。

E17. 假设给定两个序列，这两个序列假设是二叉树的前序和中序遍历序列。证明惟一地重构这棵二叉树是可能的。

E18. 对中序和后序遍历，证明或者反驳(找到一个反例)类似的结果。

E19. 对前序和后序遍历，证明或者反驳类似的结果。

E20. 找到相同数据的一对序列，这对序列不可能与同一棵二叉树的前序和中序遍历相符。(提示：采用短序列；用每个序列仅含 3 个数据项完成这个练习是可能的。)

10.2 二叉查找树

＊＊＊困难的选择

考虑在链表中查找某个目标键的问题。除了每次在表上移过一个结点外，没有其他方法，因此查找表总是变成顺序查找。正如我们所知，与二分查找相比，顺序查找通常很慢。因此，假设键按顺序排列，如果我们使用顺序表和二分查找，则查找就变得更快。假设我们也经常需要在表中做改变，如插入新的元素或删除老的元素，则使用顺序表比使用链表更慢，因为顺序表里的插入和删除每次要移动许多元素，而链表仅调整一些指针即可。

这一节的关键问题是：

是否可以找到有序表的一种实现，使得能进行快速查找(就像顺序表上的二分查找)，而且能做快速的插入和删除(就如链表)？

二叉树对这个问题提供了一个优秀的解决方案。通过把有序表的元素变成二叉树的结点，我们将发现能在 $O(\log n)$ 步内找到目标键，这个时间同二分查找一样，同时我们也能得到在 $O(\log n)$ 时间内的插入和删除算法。

＊＊＊比较树

在学习二分查找时，我们通过往左(如果目标键比当前的树结点的键小)或是往右(如果目标键比当前的树结点的键大)移，画出比较树以显示二分查找的过程。这样的比较树的例子显示在图 10.1 和图 10.6 中，图 10.6 显示为一棵链式二叉树。从这些图，我们可能对这

种方法已经很清楚了，这种方法是将结点以二叉树存储，而这种二叉树具有比较树本身的结构，用链接去描述树的关系。用这种方法，能保持链式存储的优点且获得二分查找的速度。

比较树的本质特征是：往左子树移是移到较小的键，而往右子树移是移到较大的键。二叉树结点的键的这一特殊条件是下面这个重要定义的本质部分：

定义　二叉查找树是一棵二叉树，它或者为空，或者它的每个结点有一个键(在数据元素内)，而且满足下列条件：

1. 根(如果存在)的键比根的左子树中任意结点的键大。
2. 根(如果存在)的键比根的右子树中任意结点的键小。
3. 根的左右子树还是二叉查找树。

前面的两个特征描述了相对于根结点的键的次序，第三个特征将它们扩展到树的所有结点，这样我们就能继续使用二叉树的递归结构。在检查完树根后，我们将移到它的左子树或右子树，这棵子树还是一棵二叉查找树。因此我们能在这棵较小的树上再次使用同样的方法。

＊＊＊没有相同的键

由于左子树的键严格小于根的键，右子树的键严格大于根的键，因此我们写这个定义时使用的方式保证了一棵二叉查找树中没有两个元素具有相等的键。可以修改这个定义，以允许它具有相等键的元素，但这样做会使算法略微复杂。因此我们总是假设：

> 二叉查找树中没有两个元素可以具有相等的键。

对于图 10.1 和 10.6 中的树，因为在每个结点是往左移还是往右移的决定是基于查找树定义中使用的键的相同比较，所以它自然而然地也是一棵二叉查找树。

10.2.1　有序表和实现

＊＊＊3 种观点

现在开始讲述用C++方法去操纵二叉查找树，至少可以采用 3 种不同的观点：

- 将二叉查找树看作是一种新的抽象数据类型，它有自己的定义和方法；
- 由于二叉查找树是特殊种类的二叉树，因此可以将它们的方法看作是特殊种类的二叉树方法；
- 由于二叉查找树的元素中包含键，且由于它们与有序表以同样的方式用于信息检索，因此可以将二叉查找树作为抽象数据类型有序表的新的实现加以研究。

在实践中，程序设计员有时会采用这些观点中的每一个，因此我们也是这样。我们将自己的二叉查找树类说明成二叉树类的派生类，因此我们的二叉树类代表了一个独特的 ADT。然而，新的类继承了前面二叉树类的方法。以这种方式，派生类的使用强调了头两种观点，而第三种观点经常出现在二叉查找树的应用中。客户程序代码能使用我们的类去解决同样的查找和排序问题，而这些问题本来可以用有序表处理。

1. 声明

我们已经介绍了允许操作二叉树的C++声明，我们将二叉树的这种实现作为我们的二叉查找树类模板的基础。

```
template <class Record>
class Search_tree :public Binary_tree<Record>{
public:
  Error_code insert(const Record &new_data);
  Error_code remove(const Record &old_data);
  Error_code tree_search(Record &target) const;
private:          //  Add auxiliary function prototypes here.
};
```

由于二叉查找树是从二叉树类派生而来，因此我们能够将通用二叉树中已经定义的方法用于二叉查找树，这些方法包括构造函数、析构函数、clear、empty、size、height 和前序遍历 preorder、中序遍历 inorder 和后序遍历 postorder。除了普通二叉树的方法以外，二叉查找树还允许名为 insert，remove 和 tree_search 的特殊方法。

* * * Record 和 Key

我们已将术语 Record 用于 Search_tree 的模板参数以强调二叉查找树中的元素必须具有能进行比较的键，因此，Record 类具有第 7 章里描述的行为：每一个 Record 与一个 Key 相关联。这个键能用普通的比较运算符进行比较，而且，由于我们假设记录能对应到其相应的键，因此运用到记录的比较运算符也就是运用于键的比较运算符。例如，从第 7 章以来我们已使用的所有的 Record 类和 Key 类都具有这些特征。因此，我们的二叉查找树中的元素就和有序表里的那些元素是兼容的。

就如我们先前所观察到的，在测试中，Record 类和 Key 类都使用类型 int 通常是很方便的。这样，我们的测试程序能采用如下声明并运用 Binary_tree 的方法，而不必更多地考虑记录和键：Binary_tree <int> test_tree;。

10.2.2 树查找

* * * 说明

二叉查找树的第一个重要的新方法是由它们的名字而来：在链式二叉查找树上查找具有特定目标键的元素的函数。这个方法必须满足下面的说明：

* * * 说明

```
Error_code Search_tree<Record> :: tree_search(Record&target)const;
后置条件：如果树中有一个元素，它的键与 target 匹配，则参数 target 被树中相应的记录
          所替换，且返回代码 success。否则，返回代码 not_present。
```

在应用中，这个方法将经常用一个参数 target 调用，target 仅包含一个键值。这个方法将把属于任何相应的 Record 的完整数据加到 target。

1. 策略

为了查找目标，我们首先将它与树根的元素进行比较。如果它们的键匹配，则完成查找。否则，适当地选择往左子树或往右子树，并在那棵子树中重复查找过程。

比如，在图 10.1 和 10.6 的二叉查找树中查找名字 Kim。我们首先将 Kim 和根元素 Jim 进行比较。因为 Kim 在字母表中位于 Jim 之后，所以移到右边并在下一步比较 Kim 与 Ron。由于 Kim 在 Ron 之前，所以移到左边并比较 Kim 和 Kay。现在 Kim 在 Kay 之后，因此移到右边并找到了希望的目标。

这明显是一个递归过程，因此我们将通过调用一个辅助递归函数去实现它。递归查找的终止条件将是什么？显然，如果我们找到了目标，函数将成功完成。如果没有找到，则函数将继续查找，直到到达一棵空的子树，在这种情况下，查找失败。

辅助查找函数将返回一个指向目标结点的指针给调用程序，虽然返回的指针能够用于访问存储在树对象中的数据，而能调用辅助查找的惟一函数必须是树的方法，因为只有这些方法能够将根作为参数传递。这样，从辅助函数返回结点指针将不会损害树的封装。我们得到辅助查找函数的下列说明：

* * * 说明

```
Binary_node<Record> * Search_tree<Record> :: search_for_node(
Binary_node<Record> * sub_root, const Record & target) const;
```

前置条件：sub_root 或者是 NULL，或者指向 Search_tree 的子树。

后置条件：如果 target 的键不在于树中，则返回结果 NULL，否则，返回指向包含目标的子树结点的指针。

2. 递归版

编写查找函数的最简单的方法是采用递归：

```
template <class Record>
Binary_node<Record>*Search_tree<Record>::search_for_node(
  Binary_node<Record>*sub_root ,const Record &target)const
{
  if (sub_root==NULL || sub root->data==target)return sub_root;
  else if (sub_root->data<target)
    return search_for _node(sub_root->right,target);
  else return search_for_node(sub_root->left,target);
}
```

3. 递归删除

* * * 尾部递归

出现在此函数中的递归仅仅是作为一个尾部递归，也就是说，它是这个函数里最后执行的语句。通过使用一个循环，通常能将尾部递归转换成迭代。在这个例子里，我们需要编写一个循环去替换第一个 if 语句，并且需要修改参数 root 以在树中移动。

* * *非递归的树查找

```
template <class Record>
Binary_node<Record>*Search_tree<Record>::search_for_node(
  Binary_node<Record>*sub_root,const Record &target)const
{
  while (sub_root!=NULL && sub_root->data!=target)
    if (sub_root->data<target)sub_root=sub_root->right;
    else sub_root=sub_root->left;
  return sub_root;
}
```

4. tree_search 方法

tree_search 方法只是调用辅助函数 search_for_node 去查找二叉查找树中包含与特定键相匹配的结点，然后它从树中抽取记录的副本并向用户返回一个合适的 Error_code。

```
template <class Record>
Error_code Search_tree<Record>::tree_search(Record &target)const
/*Post: If there is an entry in the tree whose key matches that in target, the parameter
        target is replaced by the corresponding record from the tree and a code of
        success is returned. Otherwise a code of not_present is returned.
  Uses: function search_for_node*/
{
  Error_code result=success;
  Binary_node<Record>*found=search_for_node(root,target);
  if (found==NULL)
    result=not_present;
  else
    target=found->data;
  return result;
}
```

5. 算法性能

我们回忆一下，tree_search 在很大程度上基于二分查找。如果将二分查找应用到有序表上并画出它的比较树，我们将看到，如果将 tree_search 应用于与这棵比较树同样的树，则二分查找恰与 tree_search 进行同样多的比较。从 7.4 节我们已经知道，二分查找对于长度为 n 的表要进行 $O(\log n)$ 次比较，这个性能相比于其他方法是优越的，因为随 n 的增加，$\log n$ 增长很慢。

* * *例子

作为一个例子，假设我们将二分查找应用于 7 个字母的表 a,b,c,d,e,f,g。结果得到的树显示在图 10.8 的(a)部分。如果将 tree_search 应用到这棵树，它将与二分查找进行同样次数的比较。

然而很可能，同样的字母会产生形状明显不同的二叉查找树，如显示在图 10.8 剩余部分中的任何一棵树。

* * *优化树

图 10.8(a)部分显示的树对查找来说可能是最佳的，它最为“茂密”：对于给定数目的顶

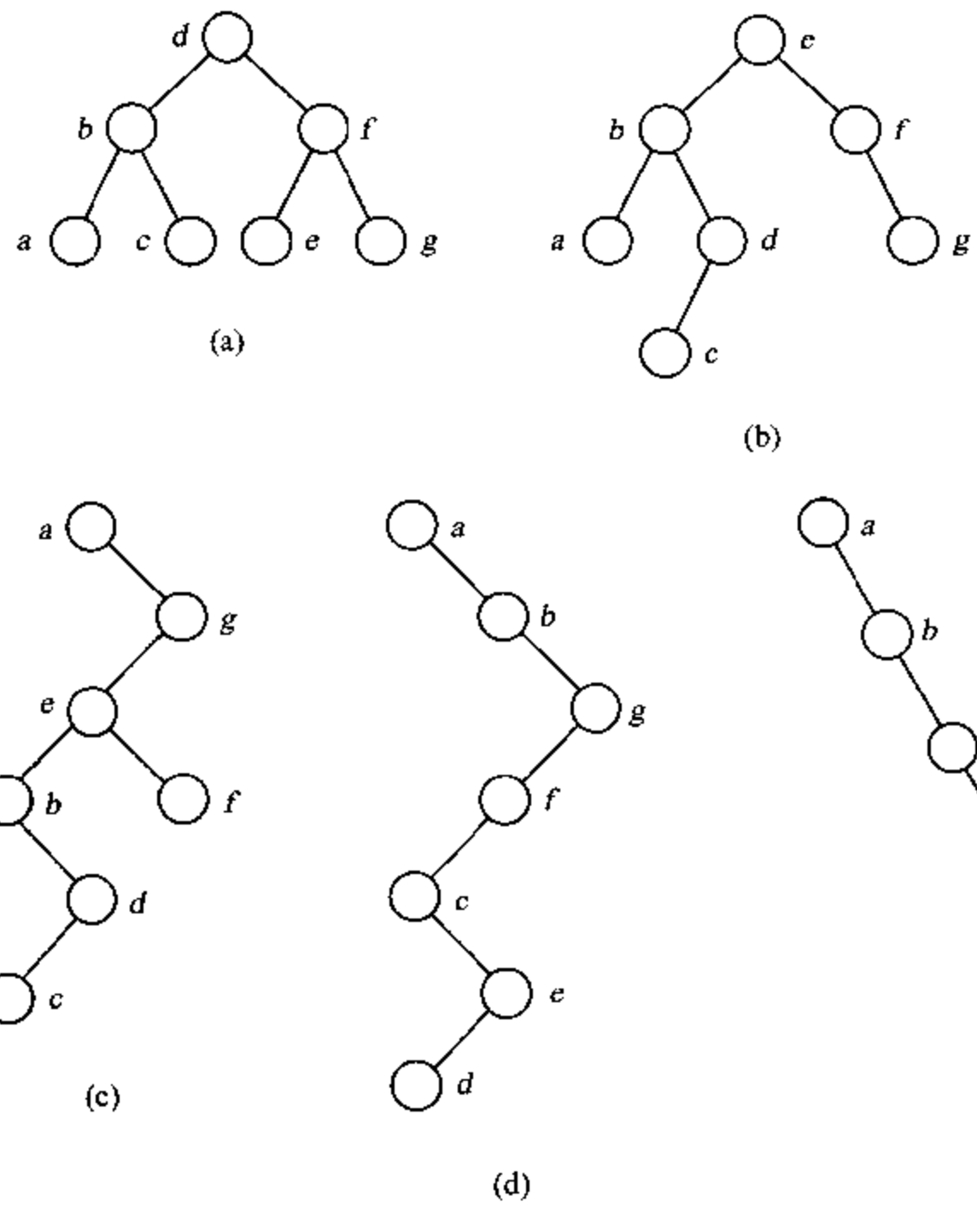


图 10.8　具有相同键的二叉查找树

点，它具有最小的高度。根与目标之间的顶点数，包括根和目标，是找到目标必须要进行的比较次数。因此，树越茂密，通常需要进行的比较次数就越少。

＊＊＊典型树

二叉查找树的形状并不总是可能预测(事先建立)，图 10.8(b)部分显示的树比(a)部分显示的树更典型，在(b)部分的树中，搜索目标 c 需要进行 4 次比较，但在(a)部分只需要进行 3 次比较。然而，(b)部分的树仍旧是相当茂密的，并且它的性能仅仅比(a)部分的优化树稍次一点。

＊＊＊次树

＊＊＊链：顺序查找

然而在图 10.8(c)部分，树退化得很恶劣，因此查找目标 c 需要 6 次比较。在(d)和(e)部分，树退化成单一的链。当将 tree_search 应用到这样的链时，tree_search 除了逐个元素地对表进行检查之外什么也不能做。换句话说，当 tree_search 应用到这样的链时，它就退化成顺序查找。对 n 个结点的树的最坏情况，tree_search 可能需要 n 次比较才能找到目标。

在实践中，如果键以随机次序进入二叉查找树，则极不可能出现退化得如图 10.8(d)和

(e)所显示的那样差的二叉查找树，相反，像(a)和(b)部分的树更加可能。因此 tree_search 的执行几乎总是同二分查找一样好。事实上，在 10.2.4 节，我们将看到，对于随机二叉查找树，tree_search 的性能仅仅比最优的 $\lg n$ 次键比较慢 39%，因此远远优于顺序查找所需要的 n 次键比较。

10.2.3 二叉查找树的插入

1. 问题

我们要考虑的下一个重要操作是将一个新结点插入到一棵二叉查找树中，使得所有的键仍保持恰当的次序，也说是说，使得结果得到的树仍满足二叉查找树的定义。这个操作的形式化说明是：

* * * 说明

```
Error_code Search_tree <Record> :: insert(const Record &new_data);
```

后置条件：如果在 search_tree 中有一个 Record 具有与 new_data 相匹配的键，则返回代码 duplicate_error。否则，Record new_data 被插入到树中，而且仍然保持二叉查找树的特征，并且返回代码 success。

2. 例子

在着手编写这个函数前，我们先研究一些简单的例子。图 10.9 显示当我们将键 e,b,d,f,a,g,c 以给定的次序插入到初始为空的树中时的情形。

当插入第一个元素 e 时，它成为根，如(a)部分所示。由于 b 在 e 之前，因此将 b 插入到 e 的左子树中，如(b)显示。接下来，我们插入 d，首先将它和 e 进行比较，结果是往左，然后将它和 b 进行比较，结果是往右。下一步我们将 f 插入到根的右边，如图 10.9(d)部分所示。由于 a 是迄今最小的键，因此它将移到 e 的左边，然后再到 b 的左边。类似地，键 g 按字母表顺序是最后一个，所以将它插入到最右边，如(f)所示。最后，在插入 c 时，首先将它和 e 进行比较，结果是往左，然后再到 b 的右边，之后是到 d 的左边。因此我们获得图 10.9 最后部分显示的二叉查找树。

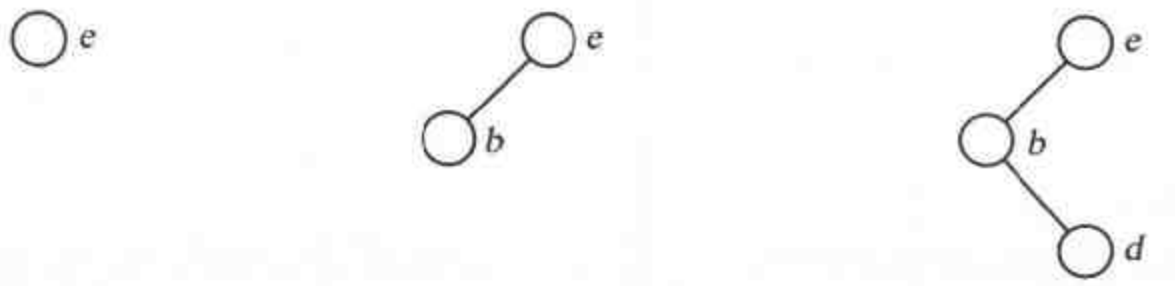
* * * 不同的次序，相同的树

很可能不同次序的插入能生成同样的二叉查找树。例如，我们可以通过插入如下两种次序之一或者其他次序的键而得到图 10.9 中最后的树：

$$e,f,g,b,a,d,c \text{ 或 } e,b,d,c,a,f,g$$

* * * 自然次序

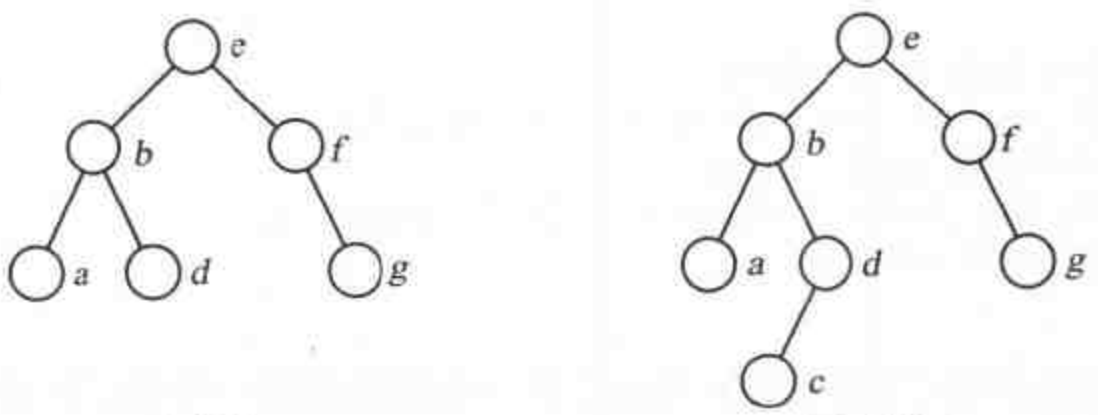
有一种情况特别重要。假设键是以自然次序 $a,b,\cdots\cdots,g$ 插入初始为空的树中，则 a 将成为根，b 将成为它的右孩子，c 将移到 a 和 b 的右边，等等。此插入将生成一个二叉查找树链，如图 10.8 中最后部分所示。就像我们已看到的，这样的链对于查找来说是非常低效的，因此我们得出结论：



(a) 插入 *e*　　(b) 插入 *b*　　(c) 插入 *d*



(d) 插入 *f*　　(e) 插入 *a*



(f) 插入 *g*　　(g) 插入 *c*

图 10.9　二叉查找树的插入

如果键以自然次序插入初始为空的二叉查找树，则 insert 方法将生成一棵退化成低效链的树，不要将 insert 方法用于已排过序的键。

如果键已按反序排列或者当它们基本有序时也可以得到同样的结论。

3. 方法

从我们已完成的例子到向二叉查找树中插入新结点的通用方法之间只需一小步。

第一种情况是将一个结点插入到一棵空树中，这比较容易，只需要将 root 指向这个新结点。如果树不为空，则我们必须将这个结点的键和根中的键进行比较。如果它比根小，则必须将新结点插入到左子树；如果它比根大，则必须将它插入右子树。键相等的情况是一种错误。

注意到我们已经用递归描述了插入过程。在将新键和根中的键进行比较之后，我们对根的左子树或者右子树使用先前在根上使用过的完全同样的插入方法。

4. 递归函数

从这个概要，我们现在能使用这一节开始时的声明去编写出函数。像寻常一样，树方法会调用一个辅助递归函数。

```
template <class Record>
Error_code Search_tree<Record> :: insert(const Record &new_data)
```

```
{
  return search_and_insert(root,new_data);
}
```

* * *递归插入

注意:例如,当树在初始时为空时,辅助函数可能需要对 Search_tree 的根做一个永久的改动。因此,辅助函数的实现必须使用一个引用参数。

```
template <class Record>
Error_code Search_tree<Record>::search_and_insert(
           Binary_node<Record>*&sub_root,const Record &new_data)
{
  if (sub_root==NULL){
    sub_root=new Binary_node<Record>(new data);
    return success;
}
else if (new_data<sub_root->data)
  return search_and_insert(sub_root->left,new_data);
else if (new_data>sub_root->data)
  return search_and_insert(sub_root->right,new_data);
else return duplicate_error;
}
```

我们记得对二叉树首要要求之一是没有两个元素共享同一个键。相应地,函数 search_and_insert 会拒绝具有重复键的元素。

在函数 insert 中使用递归并不是必须的,因为它是尾部递归。我们把将 insert 转化为非递归形式的任务留作练习。

关于性能,insert 要进行的键比较与 tree_search 在寻找待插入键时要进行的键比较相同。insert 方法也改变一个指针,但它不移动元素或是进行任何其他花费大量空间或时间的操作。因此,insert 的性能将与 tree_search 的性能非常近似。

insert 方法通常能用 $O(\log n)$ 步去将一个新结点插入到具有 n 个结点的随机二叉查找树中。一棵随机树可能退化到使插入需要 n 步,这种情况可能发生但极少发生。然而,如果将键以有序的次序插入到一棵空树中,则这种退化情况将发生。

10.2.4 树排序

从我们对遍历二叉树的讨论可以记得,当我们以中序遍历一棵二叉查找树时,键将以有序的次序出现,其原因在于给定键左边的所有键都在它之前,而右边所有的键都在它之后。通过递归,将一次又一次地应用同样的事实,直到子树只剩下一个键。因此,中序遍历总是给出键的有序序列。

1. 过程

* * *树排序

这个观察结论是称为 treesort 的有趣的排序过程的基础,我们只是使用 insert 方法去用

待排序的元素建立一棵二叉查找树，并且使用中序遍历去得到它们的有序序列。

2. 与 Quicksort 的比较

我们简单地研究一下树排序所进行的键比较的次数。第一个结点成为二叉查找树的根，不需要进行键比较。当每一个后继结点进入时，首先将它的键与根的键比较，然后它或者进入左子树或者进入右子树。注意它与快速排序之间的相似性，在快速排序的第一阶段，每个键要与第一个支点键进行比较，然后视比较结果放入左边或右边的子表。然而在树排序中，当每个结点进来后，它会进入自己在链式结构中的最终位置上。第二个结点会成为左子树或者右子树的根(按照它的键和根键的比较结果)。从那时起，所有进入这个子树的键都会与此第二个结点进行比较。类似地，在快速排序中，一个子表中的所有键要与属于这个子表的第二个支点进行比较。按此方式继续，我们得出：

定理 10.1　树排序同选择每个子表中的第一个键作为子表支点的快速排序正好具有相同的键比较次数。

* * * 优点

正如我们所知，通常快速排序是一种优秀的方法。在我们所学过的方法中，平均起来只有归并排序进行较少的键比较。因此就键比较次数来说，平均起来，我们能期待树排序也是一种优秀的排序方法。实际上，从 8.8.4 节可以得出结论：

推论 10.2　在一个长度为 n 的随机序列表中，平均起来，树排序执行的键比较次数为：

$$2n\ln n+O(n)\approx 1.39n\lg n+O(n)$$

树排序有一个优于快速排序的方面。快速排序在整个排序过程中都需要访问所有待排序的元素，而在树排序中，不必在排序过程的一开始就用到全部结点，而是当它们可用时，再一个个地构建到树中，因此树排序对于每次接收一个结点的应用更为可取。树排序的主要优点是它的查找树对于后期的插入和删除仍然可用，而且随后能以对数时间搜索这棵树，然而我们先前的所有排序方法或者需要顺序表，因此造成插入和删除困难，或者会产生只能用于顺序查找的链表。

* * * 缺点

树排序的主要缺点已隐含在定理 10.1 中。快速排序在其最坏情况下有非常恶劣的性能，而虽然仔细选择支点会使得这种情况极不可能发生，但选择每个子表中的第一个键作为支点会在键已经有序时发生最坏情况。如果提供给树排序的键已经有序，则树排序也将是一个灾难——它创建的查找树将退化成一条链。因此如果键已经有序或接近有序，则不再使用树排序。

关于树排序，几乎没有其他的不能同样应用于所有链式结构的限制。对具有较小元素项的小型问题，顺序存储通常是较好的选择，而对于大型问题和大的记录，链式存储比较盛行。

10.2.5 二叉查找树的删除

在树排序的讨论中，我们作为一个优点提到了在二叉查找树中进行修改的能力。我们已经获得将新结点插入二叉查找树的算法，而且此算法能用来更新树，就像从头开始建立它一样简单。但我们尚未考虑如何从树中删除一个元素和包含它的结点。如果被删除的结点是一个树叶，则这个过程是简单的：只需要将指向待删结点的链替换为 NULL 即可。如果待删结点只有一个非空子树则过程仍然简单：只需将待删结点的双亲的链调整到指向它的非空子树的根。

* * * 删除

然而，当被删除的结点左右子树都不为空时，问题就变得略微复杂。待删结点的双亲将指向哪棵子树？对其他子树将做哪些修改？这个问题将用图 10.10 和我们要实现的方案进行说明。首先，通过移到它的左孩子然后移到它尽可能远的右孩子，我们找出中序遍历下该结点的直接前趋。（直接后继也完全一样。）这个直接前趋没有右孩子（因为我们已经移到尽可能远的右侧），所以能够毫无困难地将它从现在的位置上删除。然后可以将它放到树中假定待删除的结点先前占用的位置上，此时二叉查找树的特性将仍然满足，因为在最初的树中

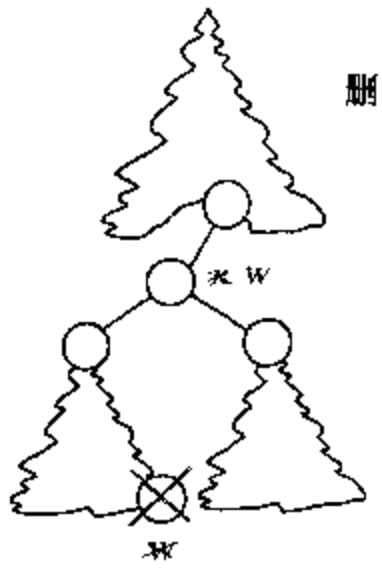


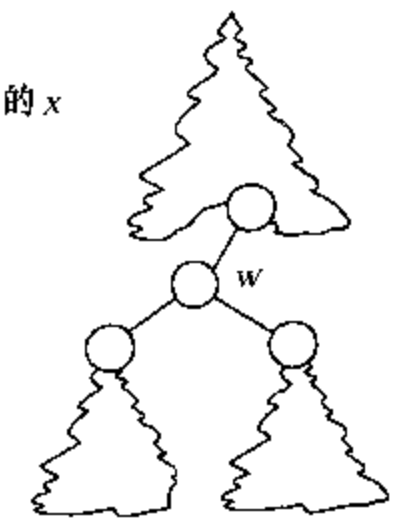
情况一：删除一个树叶



情况二：一棵子树为空







情况三：没有子树为空

图 10.10 从二叉查找树中删除一个结点

没有任何键的次序是在被删键和它的直接前趋之间。

* * * 要求

我们现在能够实现这个计划。首先考虑从二叉树中删除一个特定结点的辅助函数，这个函数使用指向待删结点的指针作为一个调用参数，而且这个参数通过引用进行传递，这样使得对它的任何修改都反映在调用环境中。由于我们的目的是去更新一棵二叉查找树，因此必须假设调用程序中的实际参数是树中的一个链，而不只是一个副本，否则树结构本身将不会按需被修改。换句话说，如果是删除 x 左边的结点，则调用将是：

remove_root(x->left)，

如果删除根，则调用将是：

remove_root(root)。

另一方面，下面的调用将不会正确工作，因为虽然它会调整指针 y，但指针 x->left 将保持不变：

y=x->left;remove_root(y);

辅助函数 remove_root 将如下实现：

* * * 删除

```
template <class Record>
Error_code Search_tree<Record>::remove_root (Binary_node<Record>
                                              *&sub_root)
/*Pre: sub_root is either NULL, or points to a subtree of the Search_tree.
  Post: If sub_root is NULL, a code of not_present is returned. Otherwise, the root of the
        subtree is removed in such a way that the properties of a binary search tree are
        preserved. The parameter sub_root is reset as the root of the modified subtree,
        and success is returned. */
{
  if (sub_root==NULL) return not_present;
  Binary_node<Record>*to_delete=sub_root;
                                              //  Remember node to delete at end.
  if (sub_root->right==NULL) sub_root=sub_root->left;
  else if (sub_root->left==NULL) sub_root=sub_root->right;
  else {                                      //  Neither subtree is empty.
    to_delete=sub_root->left;                 //  Move left to find predecessor.
    Binary_node<Record>*parent=sub_root;      //  parent of to_delete
    while (to_delete->right!=NULL){           //  to_delete is not the predecessor.
      parent=to_delete;
      to_delete=to_delete->right;
    }
    sub_root->data=to_delete->data;           //  Move from to_delete to root.
    if (parent==sub_root) sub_root->left=to_delete->left;
    else parent->right=to_delete->left;
  }
  delete to_delete;                           //  Remove to_delete from tree.
  return success;
}
```

应当跟踪一下这个函数，从而检查所有的指针都得到适当地更新，特别是不存在子树为空的情况。我们必须仔细区分根的左孩子是它的前趋这种情况和必须向右移以找到前趋这种情况。注意在后一种情况中所需的步骤是使循环在右子树为空的结点终止，但不是在空

子树本身结束。

在调用 Search_tree 的删除方法时，客户程序传递待删除的元素，而不是指向待删除的结点的指针。为了完成从树中进行这样的删除，我们将树的递归查找和前述的删除函数结合起来，结果得到的 Search_tree 方法如下。

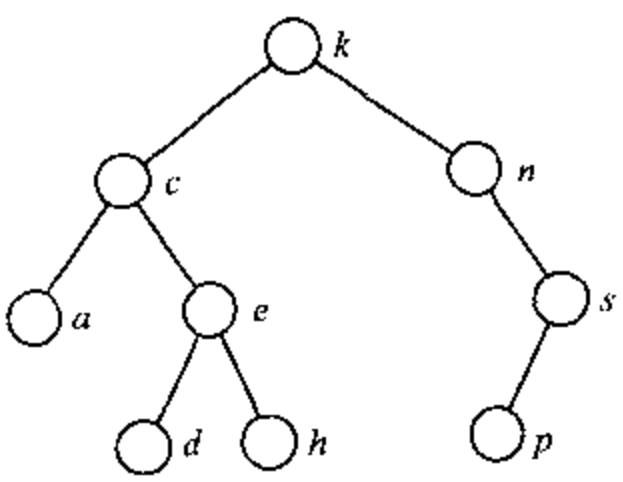
```
template <class Record>
Error_code Search_tree<Record>::remove(const Record &target)
/*Post: If a Record with a key matching that of target belongs to the Search_tree a code
        of success is returned and the corresponding node is removed from the tree.
        Otherwise, a code of not_present is returned.
  Uses: Function search_and_destroy*/
{
  return search_and_destroy(root, target);
}
```

如通常一样，这个方法使用了一个辅助递归函数，它指向树中实际的结点。

```
template <class Record>
Error_code Search_tree<Record>::search_and_destroy(
          Binary_node<Record>*&sub_root, const Record &target)
/*Pre: sub_root is either NULL or points to a subtree of the Search_tree.
  Post: If the key of target is not in the subtree, a code of not_present is returned.
        Otherwise, a code of success is returned and the subtree node containing target
        has been removed in such a way that the properties of a binary search tree have
        been preserved.
  Uses: Functions search_and_destroy recursively and remove_root*/
{
  if (sub_root==NULL || sub_root->data==target)
    return remove_root(sub_root);
  else if (target<sub_root->data)
    return search_and_destroy(sub_root->left, target);
  else
    return search_and_destroy(sub_root->right, target);
}
```

练习 10.2

前几个练习是基于下面的二叉查找树，使用初始树作为每个部分的基础，独立回答每个练习的每个部分。



E1. 说明在前述二叉查找树的每一次查找中，下列的每个目标将比较的键。

(1)c　(4)a　(7)f

(2)s　(5)d　(8)b

(3)k　(6)m　(9)t

E2. 将下列每个键插入到前述的二叉查找树中，说明每种情况中将进行的键比较。独立地完成每个部分，将键插入到原始树中。

(1)m　(3)b　(5)c

(2)f　(4)t　(6)s

E3. 使用这一节设计的算法，从前述二叉查找树中删除下列每个键。独立地完成每个部分，从原始树中删除键。

(1)a　(3)n　(5)e

(2)p　(4)s　(6)k

E4. 画出 insert 函数为 14 个名字的表构造的二叉查找树，这些名字以下列次序给出并插入到一个先前为空的二叉查找树中。

(1)Jan Guy Jon Ann Jim Eva Amy Tim Ron Kim Tom Roy Kay Dot

(2)Amy Tom Tim Ann Roy Dot Eva Ron Kim Kay Guy Jon Jan Jim

(3)Jan Jon Tim Ron Guy Ann Jim Tom Amy Eva Roy Kim Dot Kay

(4)Jon Roy Tom Eva Tim Kim Ann Ron Jan Amy Dot Guy Jim Kay

E5. 考虑建立两棵包含从 1 到 63 之间的整数键(其中包含 1 和 63)的二叉查找树，键以下列次序得到：

(1)所有的奇数按次序(1,3,5,...,63)，然后是 32,16,48，然后剩余的偶数按次序(2,4,6,...)。

(2)32,16,48，然后所有的奇数按次序(1,3,5,...,63)，然后剩余的偶数按次序(2,4,6,...)。

这些树中哪一棵能比较快地建立？解释其原因。(试着不画出树而回答这个问题。)

E6. 这个练习的所有部分引用图 10.8 显示的二叉查找树并涉及将键 a,b,...,g 插入初始为空的二叉查找树的不同次序。

(1)给出插入键的 4 种不同的次序，每一种将产生(a)部分所示的二叉查找树。

(2)给出插入键的 4 种不同的次序，每一种将产生(b)部分所示的二叉查找树。

(3)给出插入键的 4 种不同的次序，每一种将产生(c)部分所示的二叉查找树。

(4)解释为什么只有一种插入键的次序能产生退化为给定链的二叉查找树，如显示在(d)和(e)部分的二叉查找树。

E7. 递归在 insert 函数中的使用并不是必须的，因为它是尾部递归。以非递归形式重写函数 insert。(需要一个局部指针变量在树上移动。)

程序设计项目 10.2

P1. 准备包含二叉查找树定义和这一节中设计的函数的软件包，这个包应适于包含进任意的应用程序。

＊＊＊演示程序

P2. 产生一个菜单驱动的演示程序，用以说明二叉查找树的使用。元素可能由单个键组成，键应为单个字符。用户能演示的最少能力包括：构造树、插入和删除给定键的元素、查找目标键和以标准的次序去遍历树。可以通过包含这一节和前几节中作为练习编写的其他能力去增强此项目设计，这些能力包括：确定树的大小、打印出所有元素并对它们进行安排以显示出树的形状以及用各种方式去遍历树。

尽可能将函数作为模块放进你的项目中，使你以后在遇到其他的树时，能用另一种树的功能相等的软件包去替换二叉查找树的操作软件包。

＊＊＊树排序

P3. 编写树排序的函数，它能被加入 8.2 节的项目 P1 中。确定表结构是顺序的还是链式的，将其结果同第 8 章的那些排序方法进行比较。

＊＊＊监视哨查找

P4. 编写查找函数，使用如下带监视哨的二叉查找树：引进一个新的监视哨结点，并且设置一个称为 sentinel 的指针指向它，见图 10.11。用 sentinel 链（指向监视哨的链）替换二叉查找树中所有的 NULL 链，然后，对每一次查找，在开始查找前将目标键存储到监视哨结点中。从 tree_search 中删除不成功查找的测试，因为它现在不能出现，而现在找到监视哨的查找实际上是一个不成功的查找。在先前的测试数据上运行这个函数，将这个版本的性能和原始函数 tree_search 的性能进行比较。

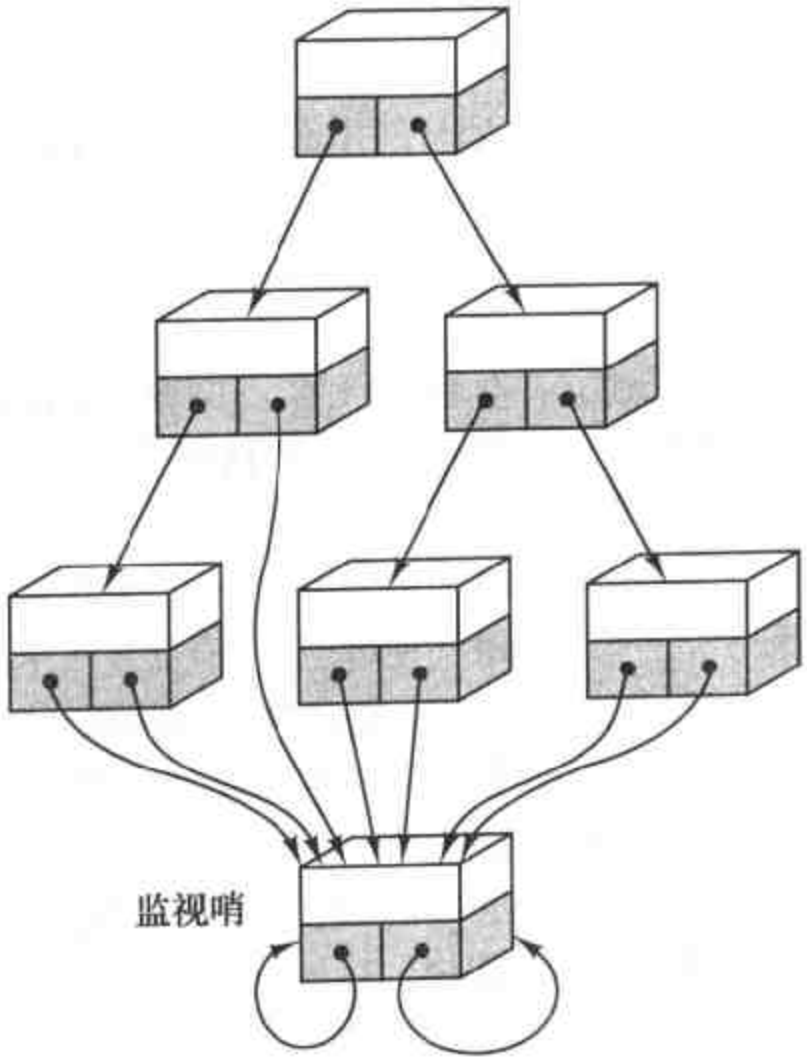


图 10.11　带监视哨的二叉查找树

＊＊＊信息检索程序

P5. 不同的作者倾向于使用不同的词且以不同频率使用通用词。给定一篇短文或其他的文

本，找出其中用了哪些不同的词及每个词各使用了多少次，这会是一个有趣的问题。本项目旨在比较对这个信息检索问题有用的几棵不同的二叉查找树。目前，项目的第一部分是产生一个驱动程序和使用普通的二叉查找树的信息检索软件包。这里给出主驱动程序的框架：

1. 创建数据结构(二叉查找树)。
2. 要求用户提供文本文件名并且打开此文件用于读取。
3. 读文件，将它分成单个的词，且将词插入到数据结构中。为每个词保留一个频率计数(这个词在输入中出现了多少次)，当遇到重复的词时，增加这个词的频率计数。相同的词仅在树中插入一次。
4. 打印出第 3 部分中进行的比较次数和使用的 CPU 时间。
5. 如果用户愿意，则以字母表顺序输出数据结构里所有的词及其频率计数。
6. 将第 2 到第 5 部分的所有内容都放入一个 do……while 循环中，这个循环将按用户的意愿多次运行。因此如果希望的话，用户能用多个文件去建立数据结构。而通过两次读取相同的文件，用户能够比较用于检索的时间和用于初始插入的时间。

以下是关于驱动程序的更多描述：

- 驱动程序的输入是一个文件，程序将以不同的文件执行多次；使用的文件名应在程序运行时从用户请求得到。
- 词定义为字母以及撇号(')、连字符(-)的序列，假设撇号或连字符都是直接放在字母前后。大小写字母将看作是一样的(通过将所有的字母转换化成小写或大写)。在文本中可能出现超过 20 个字符的词，但在本项目中将词截断为最初的 20 个字符(也即，每个词仅有 20 个字符存储在数据结构中)。非字母字符(比如数字、空格、标点符号和控制字符)在文本文件中可以出现，这些字符的任何出现将终止一个词，而且当且仅当出现一个字母时才认为出现下一个词。
- 在编写驱动程序时，确信当以后改变数据结构的实现时不用再修改程序本身。

这里首先是用二叉查找树实现的函数的说明。

```
void update(const String &word, Search_tree <Record>  &structure, int
&num_comps);
```

后置条件：如果 word 在 structure 中不存在，则将 word 插入到 structure 中，并且它的频率计数为 1。如果 word 在 structure 中已存在，则将它的频率计数加 1。变量参数 num_comps 设置为所进行的词比较次数。

```
void print(const Search_tree <Record>  &structure);
```

后置条件：在终端上以字母顺序打印出 structure 中所有的词和它们的频率计数。

```
void write_method();
```

后置条件：这个函数编写一个短字符串，标识用于 structure 的抽象数据类型。

10.3 建立二叉查找树

假设现在有一个数据已按序排列的表，或许是一个键已按字母顺序排序的记录文件，如果我们希望使用此数据去查找信息、加进额外的信息或进行其他的修改，则我们打算利用表或文件去生成一棵二叉查找树。

* * * 目标

当然，我们会从一个空二叉树开始，简单地使用树插入算法将每个元素插入到树中。但给定的元素已排过序，所以结果得到的查找树将成为一条长链，使用它将会太慢——因为此时是以顺序查找的速度而不是二分查找的速度。因此，我们希望利用这些元素并将它们建成尽可能浓密的树，使得同时减少建树所需的时间和随后所有查找所需的时间。例如，当元素数 n 是 31 时，我们希望建立如图 10.12 中的树。

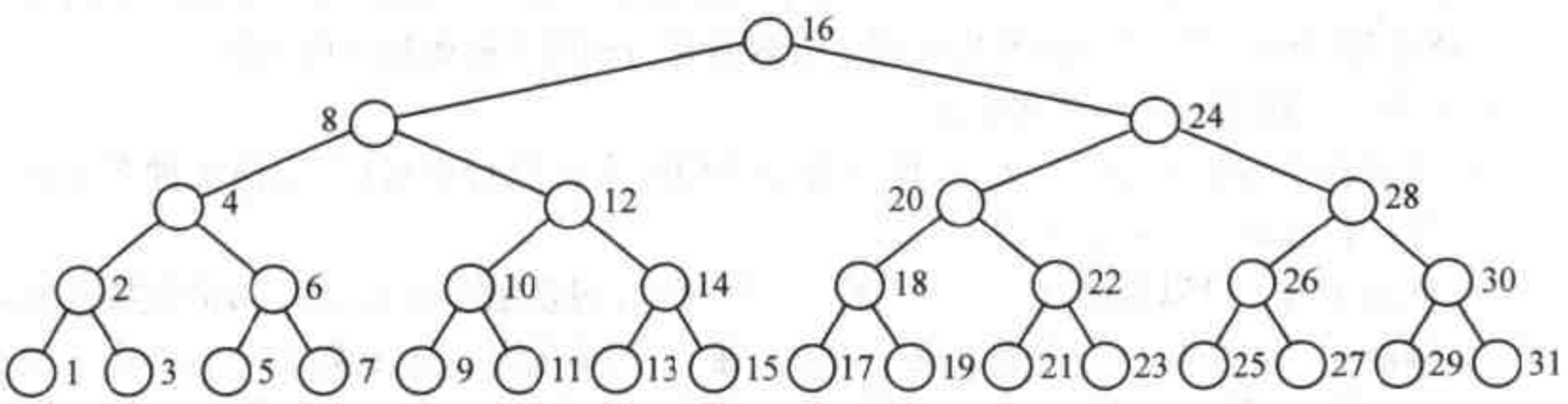


图 10.12 具有 31 个结点的完全二叉树

在图 10.12 中，元素按自然次序编号，即按中序次序编号，因为它们按有序次序被接收，因此这是接收它们并在树中建立它们的次序。我们也会使用这个编号去标记树的结点。

如果花点时间去检查这个图，可以注意到标记的一个重要特征。树叶的标记都是奇数，也就是不能被 2 整除的数。树叶上面一层的结点标记是 2,6,10,14,18,22,26 和 30，这些数都是奇数乘以 2，也就是说，它们都是偶数，但是它们都不能被 4 整除。在更上一层，标记是 4,12,20 和 28，这些数被 4 整除，但它们不能被 8 整除。最后，刚好在根下面的结点标记为 8 和 24，根是 16。关键的观察结果是：

* * * 关键特性

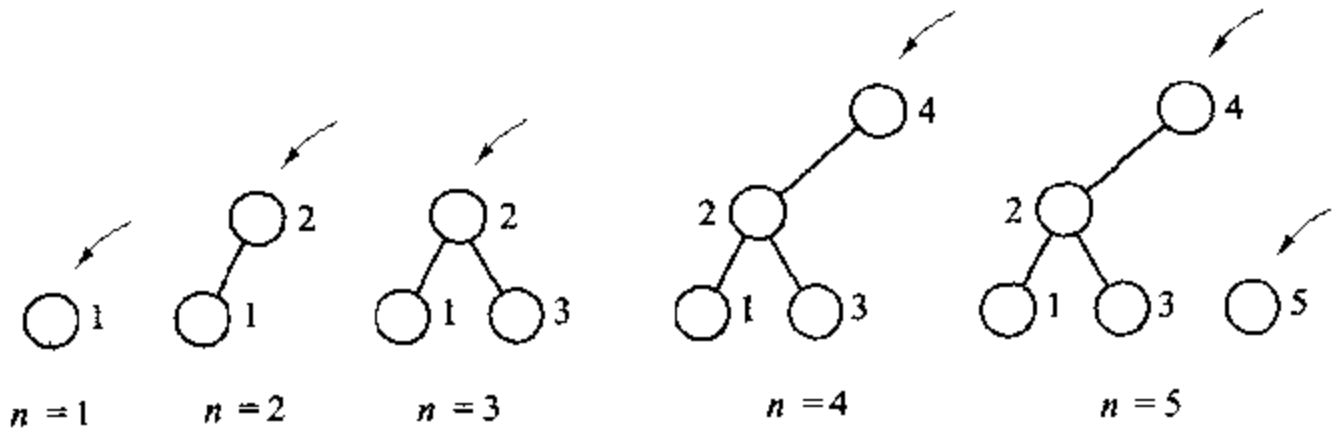
如果一棵完全二叉树的结点按中序序列从 1 开始编号，则每个结点在树叶以上的层次数恰恰等于除尽其标记的 2 的最高次幂。

现在对我们的问题再加上一个限制：假设我们事先并不知道有多少元素会被嵌入到树中。如果元素来自于一个文件或一个链表，则这个假设非常合理，因为在得到它们之前我们没有方便的方法去计算元素的个数。

这个假设也有一个好处，就是它将免除我们担心以下事实：即元素数恰好比 2 的幂小 1，从而使得结果得到的树将不是完全的且不会像图 10.12 中的树那样对称。相反，我们将设计自己的算法，仿佛它是完全对称的，并且在得到所有元素后，再确定如何整理这棵树。

10.3.1　开始

当编号为 1 的元素到达时，所要做的工作是毫无疑问的，它将被放入一个叶结点，这个叶结点的左右指针都设置为 NULL。编号为 2 的结点在结点 1 的上面，如图 10.13 所示。由于结点 2 链到结点 1，显然我们必须采用某种方式去记住结点 1 的位置，直到元素 2 到达。结点 3 又是一个树叶，但它在结点 2 的右子树中，因此我们必须记住指向结点 2 的指针。



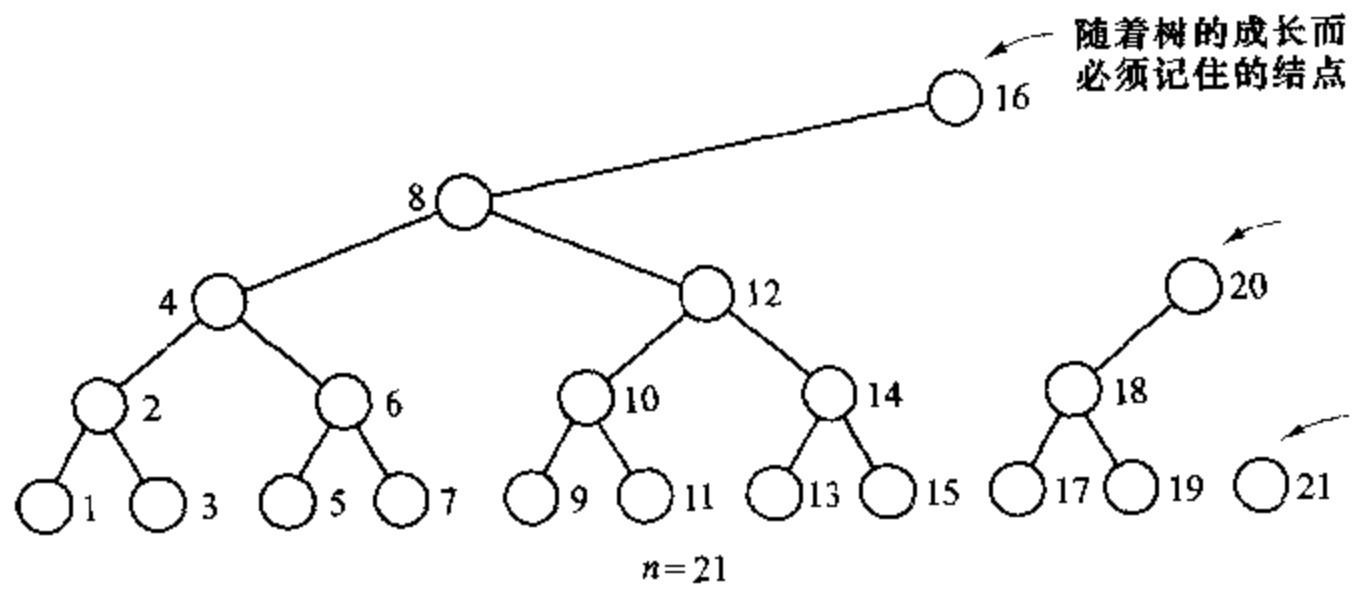


图 10.13　利用结点建立二叉查找树

这是否意味着我们必须保留一个指针列表以指向先前处理过的所有结点，从而能够决定下一步如何链接？其答案是否定的，因为当加入结点 2 的时候，结点 1 的所有连接都已完成。为了从结点 4 建立左链，必须记住结点 2，直到加入结点 4，但随后将不再需要指向结点 2 的指针。类似地，必须记住结点 4，直到已处理完结点 8。在图 10.13 中，浅灰色的箭头指向随着树的成长而必须记住的每个结点。

现在很明显，为了建立将来的链，我们仅需要记住指向每一层中最后被处理的那个结点的指针。我们将这些指针保存在一个称为 last_node 的 List 中，last_node 将非常小，例如，一棵具有 20 层(因此在 last_node 中有 20 个元素)的树能容纳 $2^{20}-1>1\ 000\ 000$ 个结点。

当加入每个新结点时，这个新结点显然是序列中最后接收的那个结点，因此我们能够将它的右指针设置为 NULL(至少临时地)。新结点如果是树叶，则它的左指针也是 NULL。否则，新结点的左指针是在 last_node 中比新结点低一层的元素。为了能同等地看待树叶和其他结点，我们认为树叶的层次为 1，并且在 last_node 的位置 0 建立初始元素，它的指针值

永远为 NULL，这个约定意味着我们将总是计算包括树叶在内的层数，因此树叶本身是在第 1 层，等等。

10.3.2 声明与主函数

我们现在能够写出自己的任务所需要的变量的声明。首先注意到，当建立一棵树时，为了创建适当的链，需要访问树的内部结构。因此，新的函数将作为一个类方法实现。而且，它将用于 Search_tree 对象，因此它将是查找树类的一个方法。于是我们将创建一个称为 Buildable_tree 的新类，这个类从 Search_tree 类派生而来，并且拥有一个新的方法，即函数 build_tree。可建立的树的说明如下：

```
template <class Record>
class Buildable_tree :public Search_tree<Record>{
public:
  Error_code build_tree(const List<Record>&supply);
private:                          //  Add auxiliary function prototypes here.
};
```

build_tree 的第一步将是接收元素。为简单起见，假设可以在一个称为 supply 的 Record 数据的 List 中找到这些元素。然而，重写这些函数从 Queue 或文件或甚至是另一个我们希望重新平衡的 Search_tree 接收数据是一件简单的事情。

当接收待插入到树中的新元素时，我们更新 count 变量以记录已加入了多少个元素。为了抽取 List supply 中的数据，显然需要 count 的值。更重要的是，count 的值决定了将容纳新元素的树的层次，因此必须将它传递给任何需要计算这个层次的函数。

当 List supply 的所有元素都插入到新的二叉查找树之后，我们必须找到树的根，然后再连接任何可能悬浮的右子树(见图 10.13 中 5 或 21 个结点的例子)。

函数因此变成：

```
template <class Record>
Error_code Buildable_tree<Record>::build_tree(const List<Record>&supply)
/*Post: If the entries of supply are in increasing order, a code of success is returned
        and the Search_tree is built out of these entries as a balanced tree. Otherwise,
        a code of fail is returned and a balanced tree is constructed from the longest
        increasing sequence of entries at the start of supply.
  Uses: The methods of class List and the functions build_insert, connect_subtrees, and
        find_root*/
{
  Error_code ordered_data=success;      //  Set this to fail if keys do not increase.
  int count=0;                          //  number of entries inserted so far
  Record x, last_x;
  List<Binary_node<Record>*>last_node;
                                        //  pointers to last nodes on each level
  Binary_node<Record>*none=NULL;
  last_node.insert(0, none);            //  permanently NULL (for children of leaves)
  while (supply.retrieve(count, x)==success){
    if (count>0&&x<=last_x){
      ordered_data=fail;
      break;
```

```
    }
    build_insert(++count, x, last_node);
    last_x=x;
  }
  root=find_root(last_node);
  connect_trees(last_node);
  return ordered_data;          //  Report any data-ordering problems back to client.
}
```

10.3.3　插入结点

前一节中的讨论说明如何正确地建立每个结点的左链，但是迄今为止所设计的过程在完成时，仍有一些结点具有必须要修改的 NULL 右链。当一个新结点到达时，它不能有非空的右子树，因为它是迄今(按次序)所接收的最后一个结点。另一方面，它可能变成某个具有较大键的结点的左孩子，在这个情况中，它的双亲结点仍未到达。通过查看 last_node 表，我们可以区分发生了哪种情况。如果 level 表示新结点在树叶之上的层次(包含树叶)，则它的双亲是在 level+1 层。我们查看位置 level+1 中 last_node 的元素，如果它的右链仍是 NULL，则它的右孩子必须是新结点，否则，它的右孩子已经到达，且新结点必须是某个未来结点的左孩子。

我们现在编写函数去将一个新结点插入到树中。

```
template <class Record>
void Buildable_tree<Record>::build_insert(int count,
                                          const Record &new_data,
                                          List<Binary_node<Record>*> &last_node)
/*Post: A new node, containing the Record new_data , has been inserted as the rightmost
        node of a partially completed binary search tree. The level of this new node is
        one more than the highest power of 2 that divides count.
  Uses: Methods of class List*/
{
  int level;            //  level of new node above the leaves,counting inclusively
  for (level=1;count % 2==0;level++)
    count/=2;           //  Use count to calculate level of next_node.
  Binary_node<Record>*next_node=new Binary_node<Record>(new data),
                     *parent;   //  one level higher in last_node
  last_node.retrieve(level-1,next_node->left);
  if (last_node.size()<=level)
    last_node.insert(level,next_node);
  else
    last_node.replace(level,next_node);
  if (last_node.retrieve(level+1, parent)==success && parent->right==NULL)
    parent->right=next_node;
}
```

10.3.4　完成任务

找到树根的任务很简单：根是树中最高的结点，因此，它的指针是 List last_node 的最后

元素。例如,图 10.13 显示了 $n=21$ 的部分树,在第 5 层有最高的结点 16,而且它将是最终完成的树的根。指向每一层遇到的最后结点的指针存储在表 last_node 中,如图 10.14 显示。

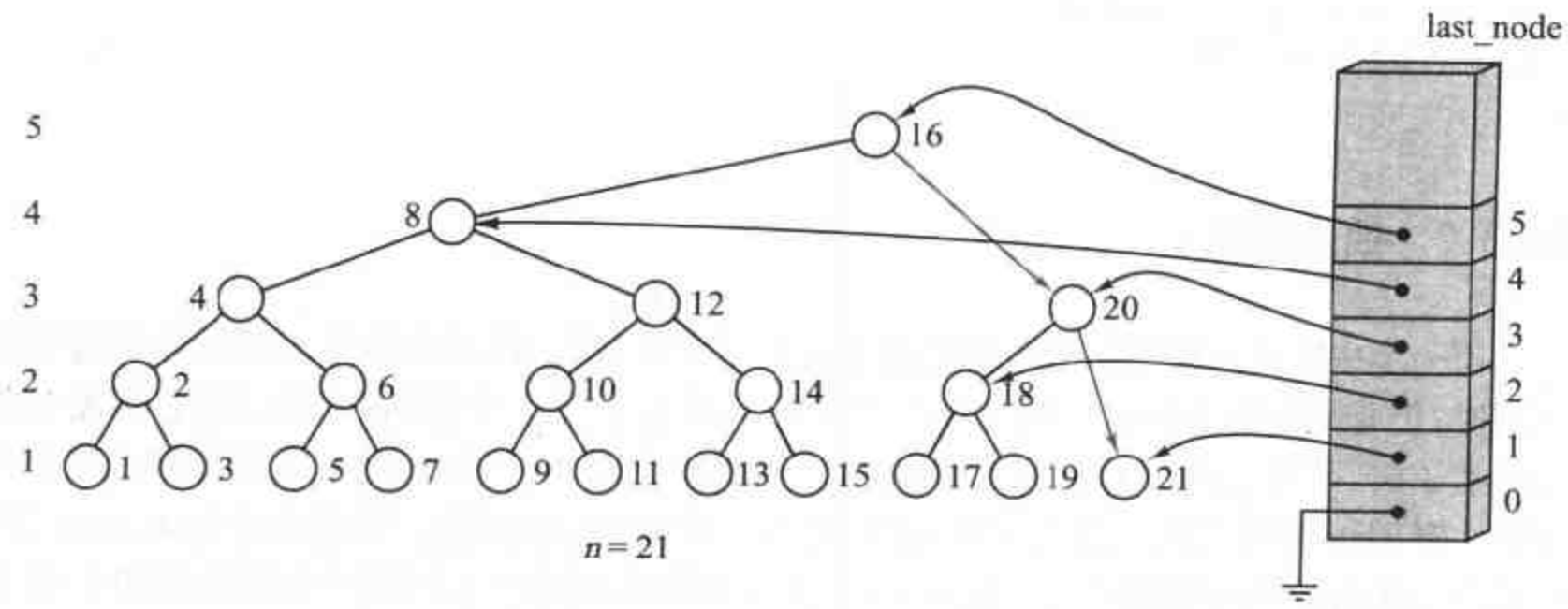


图 10.14 完成二叉查找树

我们因此得到如下函数:

```
* * * 查找根

template <class Record>
Binary_node<Record>*Buildable_tree<Record>::find+root(
                    List<Binary_node<Record>*> &last_node)
/*Pre: The list last_node contains pointers to the last node on each occupied level of
       the binary search tree.
  Post:A pointer to the root of the newly created binary search tree is returned.
  Uses:Methods of class List*/
{
  Binary_node<Record>*high_node;
  last_node.retrieve(last_node.size()-1,high_node);
    //  Find root in the highest occupied level in last_node.
  return high_node;
}
```

最后,在接收完所有的结点后,我们必须决定如何捆绑可能还未正确连接的任何子树。例如,如果 $n=21$,则我们必须将图 10.13 中显示的 3 个部分连接到单一的一棵树中。按程序设计术语,问题是树上部的一些结点仍可能使自己的右链设为 NULL,即使有更多已插入的结点属于它们的右子树。

这些结点中的任何一个(非叶结点,它的右子树仍为 NULL)将是 last_node 中的结点之一。对 $n=21$,这些结点将是结点 16 和 20(分别位于 last_node 的位置 5 和 3),如图 10.14 所示。

在下面的函数里,使用指针 high_node 去引用右子树为 NULL 的结点。我们需要确定一个指向 high_node 的右孩子的指针 lower_node,指针 lower_node 能被确定为已经不在 high_node 左子树中的 last_node 的最高结点。为了确定一个结点是否在左子树中,我们只需将它的键和 high_node 的键进行比较。

* * * 将子树捆绑到一起

```
template <class Record>
void Buildable_tree<Record>::connect trees(
  const List<Binary node<Record>*>&last_node)
/*Pre: The nearly- completed binary search tree has been initialized. The List last_
       node has been initialized and contains links to the last node on each level of
       the tree.
  Post: The final links have been added to complete the binary search tree.
  Uses: Methods of class List*/
{
  Binary_node<Record>*high_node,     //  from last_node with NULL right child
                     *low_node;
                          //  candidate for right child of high_node
  int high_level=last_node.size()-1,
    low_level;
  while (high_level>2){      //  Nodes on levels 1 and 2 are already OK.
    last_node.retrieve(high_level, high_node);
    if (high_node->right!=NULL)
      high_level--;          //  Search down for highest dangling node.
    else{                    //  Case: undefined right tree
      low_level=high_level;
      do {                   //  Find the highest entry not in the left subtree.
        last_node.retrieve(--low_level,low_node);
      }while (low_node!=NULL &&low_node->data<high_node->data);
      high_node->right=low_node;
      high_level=low_level;
    }
  }
}
```

10.3.5　评价

这一节的算法产生并非总是完全平衡的二叉查找树，例如图 10.14 中具有 21 个结点的树。如果树有 31 个结点，则它将完全平衡，但如果进来 32 个结点，则结点 32 将变成树的根，并且所有剩余的 31 个结点将是它的左子树。在这个例子里，从根删除树叶需要 5 步。如果根经过优化选择，则从根到大多数树叶只需 4 步，只有一个树叶需要 5 步。因此，在 32 个结点的树中，通常比必要的比较多做一次。

在二分查找中，一次额外的比较实际上并不需要付出很高的代价，很容易看出，我们的方法产生的树离理想的树从不会超过一层。存在一些复杂的方法，它们能建立尽可能平衡的二叉查找树，但大多数情况下仍是推荐采用更简单的方法，其中一个原因是不需要事先知道树中有多少结点。

练习中略述了采用我们的算法去重排任意的二叉查找树的结点以使它更加平衡，从而改善查找时间。同样，存在一些更复杂(然而可能更慢)的方法用于重新平衡一棵树。在 10.4 节中，我们将学习 AVL 树，并在始终保持树处于接近平衡状态的情况下，对 AVL 树进行插入和删除。然而，为了诸多实践目的，这一节描述的较简单的方法将证明已够用。

10.3.6 随机查找和优化

为了结束这一节，我们要问问，平均来说，是否值得去保持二叉查找树平衡或者去重新平衡二叉查找树。如果假设键是按随机次序到达，则平均起来，在结果得到的树中进行一次查找所需要的键比较比完全二叉树中所需要的键比较多多少次？

＊＊＊扩展二叉树

为了回答这个问题，我们先将二叉查找树如下转换为 2-树。考虑二叉树中画作圆圈的所有顶点，并且加入新的、方格顶点以代替所有空的子树(NULL 链)。这个过程显示在图 10.15 中。初始二叉树的所有顶点都变成 2-树的内部顶点，而新的顶点都是外部的(树叶)。

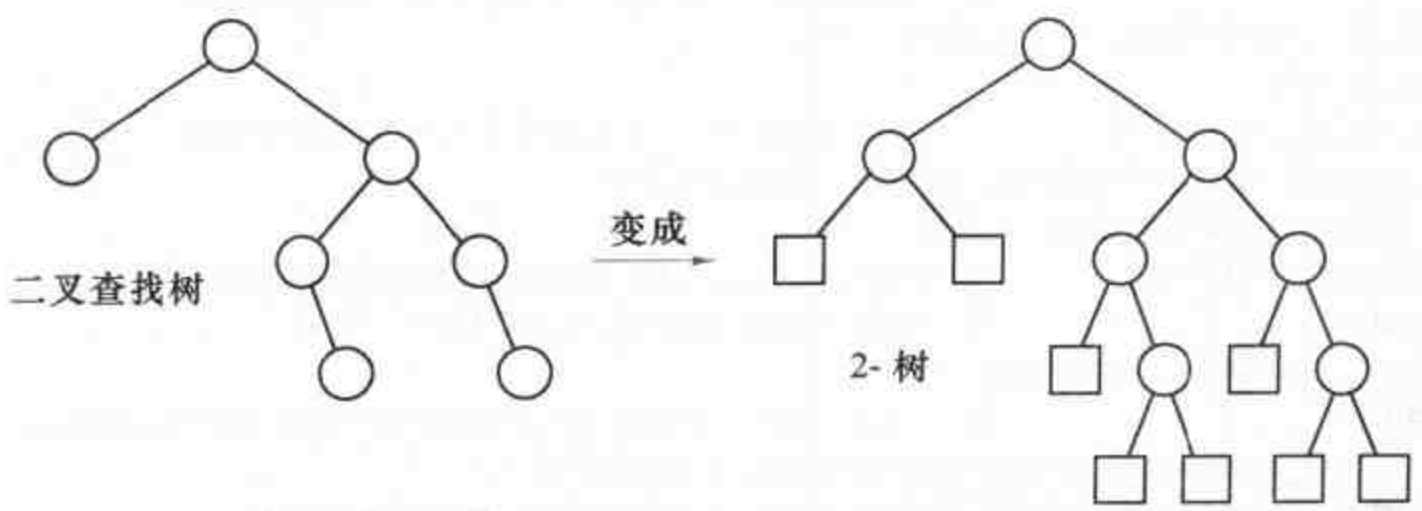


图 10.15 二叉树到 2-树的扩展

＊＊＊一一对应

我们也能以反方向应用图 10.15：从 2-树剪去所有的树叶(画作方格顶点)，它就变成了一棵二叉树。事实上，我们能说：

如果将 2-树的左和右看作互不相同，则二叉查找树和 2-树之间存在一一对应关系。

我们必须画出这种左右区别，因为在二叉查找树中，左右区别反映了键的次序，而在任意的 2-树中(没有左右区别)，分支可能表示任一 2-路决定。

＊＊＊路径长度

二叉树中成功的查找在相应 2-树的内部顶点处终止，不成功的查找在树叶处终止，因此 2-树的内部路径长度使我们得到成功查找的比较次数，2-树的外部路径长度使我们得到不成功查找的比较次数。由于在每个内部结点上要进行两次比较，因此进行一次查找，树中的每个结点进行的比较次数实际上是内部路径长度的两倍。

我们假设在建立二叉查找树时，$n!$ 种可能的键序产生的机率均等。当树中有 n 个结点时，我们用 $S(n)$表示平均成功查找进行的比较次数，用 $U(n)$表示平均不成功查找进行的比较次数。

＊＊＊计算比较次数

查找树中任意键所需要的比较次数正好比将它插入到第一个位置上所需要的比较次数多一次，并且插入它需要的比较次数与表示它不在树中的不成功查找所需要的比较次数相

同。因此，我们得到下面的关系：

$$S(n) = 1+\frac{U(0)+U(1)+\cdots+U(n-1)}{n}$$

在定理 7.4 中给出了内部和外部路径长度之间的关系，它表明：

$$S(n)=(1+\frac{1}{n})U(n)-3$$

＊＊＊递归关系

由上面两个等式一道得出$(n+1)U(n)=4n+U(0)+U(1)+\cdots+U(n-1)$。我们通过用 $n-1$ 代替 n 写出等式去解这个递归式：$nU(n-1)=4(n-1)+U(0)+U(1)+\cdots+U(n-2)$，将后将两个式子相减，得到：

$$U(n) = U(n-1)+\frac{4}{n+1}$$

＊＊＊调和数

$$H_n = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

的和称为第 n 个调和数，它在定理 A.4 中给出，这个数近似于自然对数 $\ln n$。由于 $U(0)=0$，现在能够从底部开始相加去求出 $U(n)$：

$$U(n)=4\left[\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n+1}\right]=4H_{n+1}-4\approx 4\ln n。$$

由定理 7.4，成功查找的比较次数也近似为 $4\ln n$。由于在查找任意二叉查找树时，在每个结点上需要两次比较，而且最佳的高度是 $\lg n$，因此最佳的比较次数是 $2\lg n$。但(见 A.2 节)

$$\ln n=(\ln 2)(\lg n)$$

将自然对数转换为底数为 2 的对数，我们最终得到：

定理 10.3　n 个结点的平均二叉查找树的一次查找中，访问的平均结点数近似为 $2\ln n=(2\ln 2)(\lg n)\approx 1.39\lg n$，键比较次数近似为 $4\ln n=(4\ln 2)(\lg n)\approx 2.77\lg n$。

推论 10.4　平均二叉查找树需要的比较次数近似于完全平衡树的 $2\ln 2\approx 1.39$ 倍。

＊＊＊非平衡的代价

换句话说，非平衡二叉查找树的平均代价是多出近 39%的比较。在优化显得比较重要的应用中，必须在此代价和平衡这棵树或维持这棵树平衡所需要的额外代价之间做出权衡。特别需要注意的是，后者的任务不仅包括计算机时间代价，而且也包括额外程序设计所需要的代价。

练习 10.3

E1. 对下列的 n 值，(如图 10.13)画出使用本节方法构造的部分二叉查找树的序列：(1)6，7，8；(2)15，16，17；(3)22，23，24；(4)31，32，33。

E2. 当 supply 是队列时，编写函数 build_tree。

E3. 当输入结构是一棵二叉查找树时，编写函数 build_tree。（这个版本给出了重新平衡二叉查找树的函数。）

E4. 编写一个版本的 build_tree 函数，它从文件中读取键，文件的每行是一个键。（这个版本给出了从有序文件读取二叉查找树的函数。）

E5. 将图 10.8 中显示的每棵二叉查找树扩展成 2-树。

E6. 键 1,2,3 有 6=3! 种可能的有序序列，但 3 个结点只有 5 种不同的二叉树。因此，这些二叉树不是以同等的可能作为查找树出现。找出 5 棵二叉查找树的哪一棵对应于 1,2,3 的 6 种可能的有序序列的每一个，于是也就找到了从随机有序输入构建每棵二叉查找树的可能性。

E7. 4 个键 1,2,3,4 有 24=4! 种可能的有序序列，但 4 个结点仅有 14 种不同的二叉树。因此，这些二叉树不是以同等可能作为查找树出现。找出 14 棵二叉查找树的哪一棵对应于 1,2,3,4 的 24 种可能的有序序列的每一个，于是也就找到了从随机有序输入构建每棵二叉查找树的可能性。

E8. 如果 T 是任意的二叉查找树，设 $S(T)$ 表示 T 中对应于 T 的有序键序列（也就是，如果将它们以给定的次序插入到一棵初始为空的二叉查找树，则将生成 T）的个数。找出关于 $S(T)$ 的公式，此公式仅仅依赖于 L 和 R 的大小以及 $S(L)$ 和 $S(R)$，这里 L 和 R 分别是 T 的根的左子树和右子树。

10.4 高度平衡：AVL 树

10.3 节的算法能用于建立一棵接近平衡的二叉查找树，或在完全重构这棵树可行时用于恢复平衡，然而在许多应用中，插入和删除以不可预测的次序连续出现。在其中一些应用中，始终保持树非常接近平衡从而优化查找次数是重要的。这一节中实现此目标的方法是由两位俄国数学家 G. M. Adel′son—Vel′skii 和 E. M. Landis 于 1962 年描述的，为了纪念他们，结果得到的二叉查找树称为 AVL 树。

*** 接近优化的高度

AVL 树实现了这个目标，即在 n 个结点的树中查找、插入和删除都能在 $O(\log n)$ 时间内达到，即使是在最坏的情况下。n 个结点的 AVL 树的高度，就像我们将建立的树，不可能超过 $1.44 \lg n$，即使是在最坏的情况下，AVL 树的性能也不会低于随机二叉查找树许多。然而，在几乎所有的情况下，查找的实际长度非常接近 $\lg n$，因而 AVL 树的性能非常接近于理想的、完全的平衡二叉查找树。

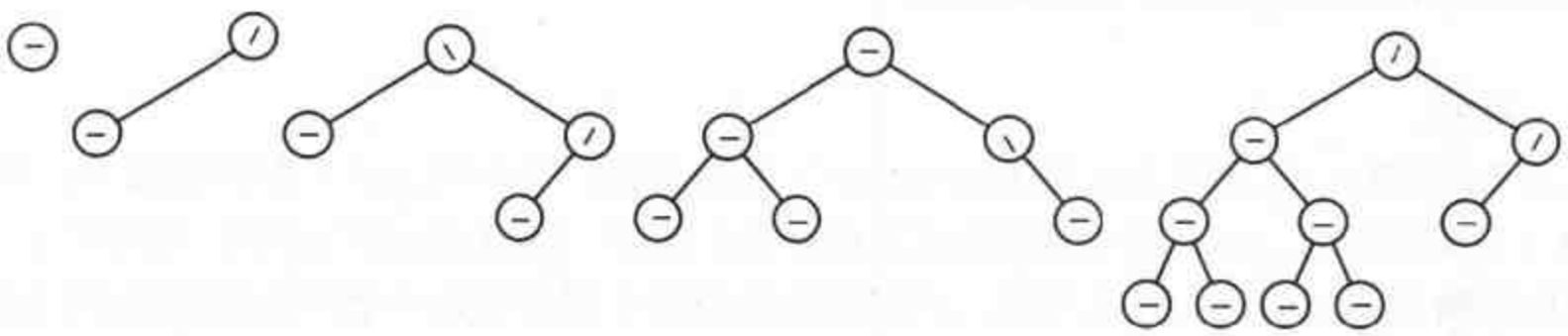
10.4.1 定义

在一棵完全平衡树中，任何结点的左右子树将有同样的高度。虽然我们不是总能达到这个目标，但通过仔细地建立查找树，我们总能保证每棵左子树和右子树的高度差从不超过 1。因此我们给出如下的定义：

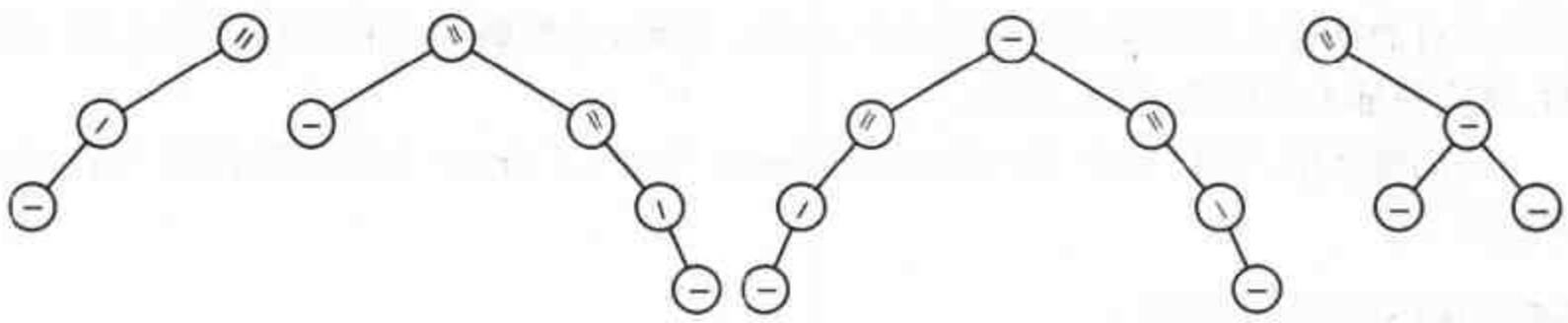
定义　AVL 树是一棵二叉查找树，它的根的左子树和右子树的高度差至多为 1，而且左右子树也是 AVL 树。

AVL 树的每个结点联系着一个平衡因子，根据左子树的高度大于、等于或小于右子树的高度，平衡因子可以为左-高、等-高或右-高(left-higher，equal-height，right-higher)。

在画图过程中，将平衡因子是左－高的结点用'/'表示，平衡因子是等－高的结点用'－'表示，平衡因子是右－高的结点用'\'表示。图 10.16 显示了几棵小的 AVL 树和一些不满足定义的二叉树。



AVL 树



非 AVL 树

图 10.16　AVL 树和其他二叉树的例子

注意到此定义未要求所有的树叶在同一层或相邻层。图 10.17 显示了几种非常歪斜的 AVL 树，它们的右子树的高度比左子树大。

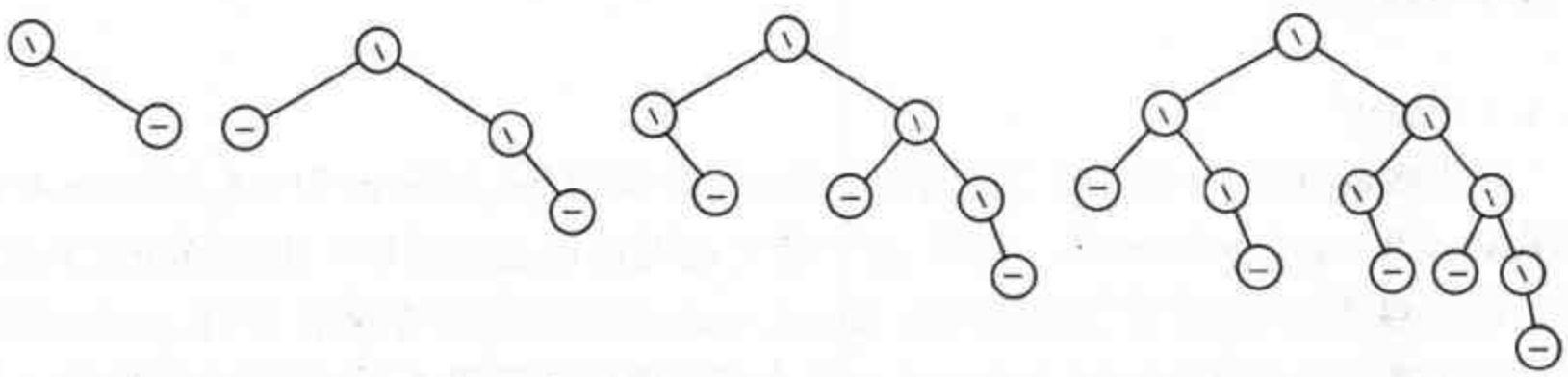


图 10.17　向右歪斜的 AVL 树

1. C++约定

我们采用枚举类型去记录平衡因子：**enum** Balance_factor {left_higher，equal_height，

right_higher };。平衡因子必须包含在 AVL 树的所有结点中，我们必须相应地改写以前的结点说明。

```
template <class Record>
struct AVL_node:public Binary_node<Record>{
//  additional data member:
  Balance_factor balance;
//  constructors:
  AVL_node();
  AVL_node(const Record &x);
//  overridden virtual functions:
  void set_balance(Balance_factor b);
  Balance_factor get_balance()const;
};
```

＊＊＊好处

这个说明的一个微妙之处是 Binary_node 的左右指针 left 和 right 的类型是 Binary_node ＊，因此，AVL_node 的继承指针成员也有这个类型。然而，在 AVL 树中，明显需要使用指向其他 AVL 结点的 AVL 结点。明显的指针类型不相容不是一个严重的问题，因为指向基类对象的指针也能指向派生类的对象。在我们的情形中，AVL_node 的 left 和 right 指针能合法地指向其他的 AVL 结点。我们用派生结构实现 AVL 结点所得到的好处是可以重用所有的处理二叉树和查找树结点的函数。然而，必须确信在将新结点插入到 AVL 树时，确实只插入真正的 AVL 结点。

我们将使用 AVL_node 的方法 get_balance 和 set_balance 去检验和调整 AVL 结点的平衡因子。

```
template <class Record>
void AVL_node<Record>::set_balance(Balance_factor b)
{
  balance=b;
}
template <class Record>
Balance_factor AVL_node<Record>::get_balance()const
{
  return balance;
}
```

＊＊＊哑方法

在大多数情况下，我们将通过指向结点的指针调用 get_balance 和 set_balance，调用形式如：left->get_balance()。然而，这个特定的调用可能会给C++编译器带来问题，C++编译器将把 left 看作仅仅是指向 Binary_node 而不是指向专用的 AVL_node，编译器因此拒绝表达式 left->get_balance()，因为它不能确定对象 ＊left 是否有这样的一个方法。我们将通过在 Binary_node 结构中包含 get_balance() 和 set_balance() 的哑版本去解决这个困难。这些哑函数作为 Binary_node 的方法被包含，单纯地供派生 AVL 树完成的实现使用。

＊＊＊虚函数

当我们将适宜的哑方法加到 struct Binary_node 后，编译器将允许表达式 left->set_

balance()。然而，编译器还有一个问题不能解决：它将使用方法的 AVL 版本还是哑版本？正确的选择仅能在运行时已知对象 * left 的类型后才能做出。因此，我们必须将 Binary_node 版本的 set_balance 和 get_balance 定义为虚方法，这意味着在运行时选择是使用哑版本还是更有用的 AVL_node 版本。例如，如果 set_balance()作为 AVL_node 的方法被调用，则将使用 AVL 版本；而如果它作为 Binary_node 的方法被调用，则将使用哑版本。

下面是 Binary_node 更改后的说明以及适当的哑方法的实现。

```
template <class Entry>
struct Binary_node{
//  data members:
  Entry data;
  Binary_node<Entry>*left;
  Binary_node<Entry>*right;
//  constructors:
  Binary_node();
  Binary_node(const Entry &x);
//  virtual methods:
  virtual void set_balance(Balance_factor b);
  virtual Balance_factor get_balance()const;
};
template <class Entry>
void Binary_node<Entry>::set_balance(Balance_factor b)
{
}
template <class Entry>
Balance_factor Binary_node<Entry>::get_balance()const
{
  recturn equal_height;
}
```

前面的任何类和函数都不需要做出其他相关的改变，所有以前的结点处理函数现在都可以用于 AVL 结点。

我们现在能说明自己的 AVL 树类。我们只需要覆盖早先的具有维护平衡树结构的插入和删除函数，其他的二叉查找树方法则可以不做任何改变地继承。因此，我们得到下面的说明：

```
template <class Record>
class AVL_tree:public Search_tree<Record>{
public:
  Error_code insert(const Record &new_data);
  Error_code remove(const Record &old_data);
private:                  //  Add auxiliary function prototypes here.
};
```

这个类的继承数据成员是指针 root，这个指针的类型为 Binary_node＜Record＞*，因此，不难看出，它能存储普通二叉树结点或 AVL 树结点的地址。我们必须保证覆盖后的 insert 方法只创建了 AVL_node 类型的结点，这样做将保证经由 AVL 树的 root 指针而到达的所有结点是 AVL 结点。

10.4.2 结点的插入

1. 引言

* * *普通的算法

* * *问题

我们首先跟随普通的二叉树插入算法将一个新结点插入到 AVL 树中：将新结点的键和根的键进行比较，并将新结点适当地插入左子树或右子树。经常出现插入新结点而不用改变树的高度的情况，在这种情况下，树的高度或根的平衡都不会改变。即使子树的高度增长，可能也是较短的子树增长了，因此只有根的平衡因子会改变。惟一可能引起困难的情况出现在将结点插入到根的严格高于其他子树的子树中，于是高度增加了。这将引起一棵子树的高度比其他子树的高度大 2，而 AVL 的条件是高度差不超过 1。在我们更仔细地考虑这个情况之前，我们先用几次插入去说明图 10.18 中 AVL 树的增长，然后用C++对我们的算法进行编码去实现这一思想。

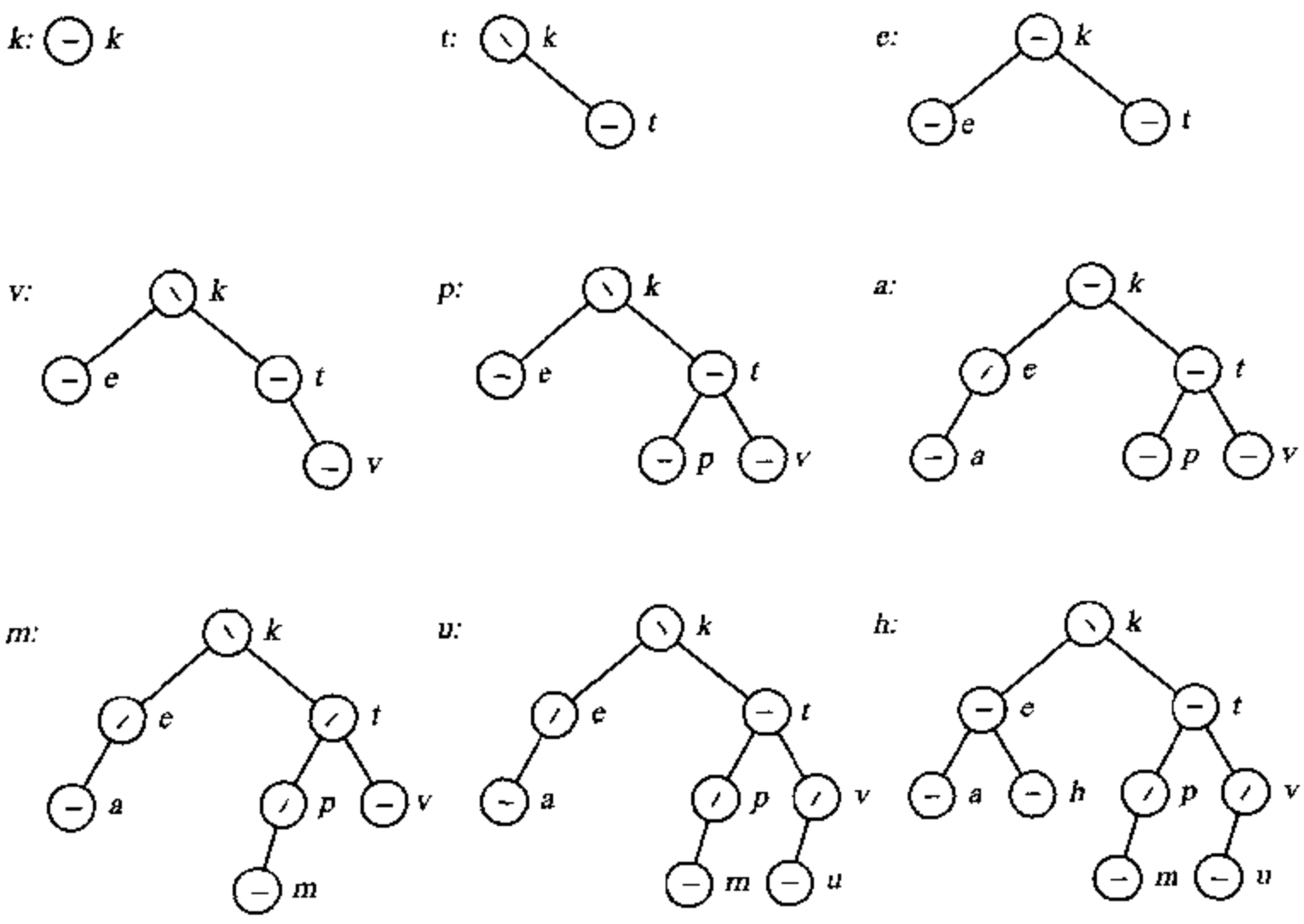


图 10.18 AVL 树中结点的简单插入

* * *例子

除了必须调整平衡因子以外，图 10.18 中的插入与普通二叉查找树中的插入以极其相似的方式进行。然而要注意的是平衡因子仅能在插入完成后才能确定。例如在图 10.18 中，当插入 v 时，根的平衡因子 k 会发生改变，但当下一次插入 p 时，k 不会改变。v 和 p 都(递归地)插入到根的右子树 t，且仅仅当插入完成后，根 k 的平衡因子才能确定。

我们算法的基本结构因此与 10.2.3 节的普通递归二叉树的插入算法一样，但具有重要的附加内容，以适应 AVL 树平衡因子和其他结构的处理。

我们必须明了（在递归后）插入是否已增加了树的高度，以致平衡因子能作出适当的改变。我们通过在由插入方法调用的辅助递归函数中增加一个类型为 bool 的附加调用参数 taller 来完成这个工作。当需要时，恢复平衡的任务将在补充函数 left_balance 和 right_balance 中完成。

有了这些决定，我们现在能编写方法和辅助函数去将新数据插入到 AVL 树中。

```
template <class Record>
Error_code AVL_tree<Record>::insert(const Record &new_data)
/*Post: If the key of new_data is already in the AVL_tree, a code of duplicate_error is
        returned. Otherwise, a code of success is returned and the Record new data is
        inserted into the tree in such a way that the properties of an AVL tree are
        preserved.
  Uses: avl_insert.*/
{
  bool taller;
  return avl_insert(root, new_data, taller);
}

template <class Record>
Error_code AVL_tree<Record>::avl_insert(Binary_node<Record>*&sub_root,
            const Record &new_data, bool &taller)
/*Pre: sub root is either NULL or points to a subtree of the AVL tree
  Post: If the key of new_data is already in the subtree, a code of duplicate_error is
        returned. Otherwise, a code of success is returned and the Record new_data is
        inserted into the subtree in such a way that the properties of an AVL tree have
        been preserved. If the subtree is increased in height, the parameter taller is
        set to true ; otherwise it is set to false .
  Uses: Methods of struct AVL_node; functions avl_insert recursively,
        left_balance, and right_balance.*/
{
  Error_code result=success;
  if (sub_root==NULL){
    sub_root=new AVL_node<Record>(new data);
    taller=true;
  }
  else if (new_data==sub_root->data){
    result=duplicate_error;
    taller=false;
  }
  else if (new_data<sub_root->data){   //  Insert in left subtree.
    result=avl_insert(sub_root->left, new_data, taller);
    if (taller==true)
      switch (sub_root->get_balance()){   //  Change balance factors.
      case left_higher:
        left_balance(sub_root);
        taller=false;          //  Rebalancing always shortens the tree.
        break;
      case equal_height:
        sub_root->set_balance(left_higher);
        break;
```

```
      case right_higher:
        sub_root->set_balance(equal_height);
        taller=false;
        break;
      }
  }
  else{          //  Insert in right subtree.
    result=avl_insert(sub_root->right,new_data,taller);
    if(taller==true)
      switch(sub_root->get_balance()){
      case left_higher:
        sub_root->set_balance(equal_height);
        taller=false;
        break;
      case equal_height:
        sub_root->set balance(right_higher);
        break;
      case right_higher:
        right_balance(sub_root);
        taller=false;        //  Rebalancing always shortens the tree.
        break;
    }
  }
  return result;
}
```

2. 旋转

我们现在考虑将新结点插入到根结点的较高子树而增加了它的高度这种情况，因此现在一棵子树比其他的子树高 2，并且这棵树不再满足 AVL 树的要求，因而必须重建部分树以恢复它的平衡。为了明确，现在假设已将新结点插入到右子树中，且原始的树是右一高，因此新结点的插入导致树的高度增加，也就是说，我们希望考虑由函数 right_balance 覆盖的情况。设 root 指示树根且 right_tree 表示它的右子树的根。

按照 right_tree 的平衡因子，需要考虑 3 种情况。

3. 情况 1:右高

*** 左旋转

第一种情况，right_tree 是右高，如图 10.19 说明。在这种情况下需要的操作称为左旋转，我们将结点 right_tree 向上旋转到 root，将 root 下降到 right_tree 的左子树中，键在 root 和 right_tree 之间的那些结点的子树 T_2 现在变成了 root 的右子树而不 right_tree 的左子树。左旋转可以简洁地描述成下面的C++函数。特别注意的是，当以合适的次序进行时，这些步骤构成了 3 个指针变量中值的旋转。同时注意，在旋转后，旋转树的高度已减 1，它是先前因为插入而增加的，所以此高度在其开始的地方结束。

```
template<class Record>
void AVL_tree<Record>::rotate_left(Binary_node<Record>*&sub_root)
/*Pre: sub_root points to a subtree of the AVL_tree. This subtree has a nonempty right
       subtree.
```

```
  Post: sub_root is reset to point to its former right child, and the former sub_root
        node is the left child of the new sub_root node.*/
{
  if (sub_root==NULL || sub_root->right==NULL)        //  impossible cases
    cout<<"WARNING: program error detected in rotate left"<<endl;
  else {
    Binary_node<Record>*right_tree=sub_root->right;
    sub_root->right=right_tree->left;
    right_tree->left=sub_root;
    sub_root=right_tree;
  }
```

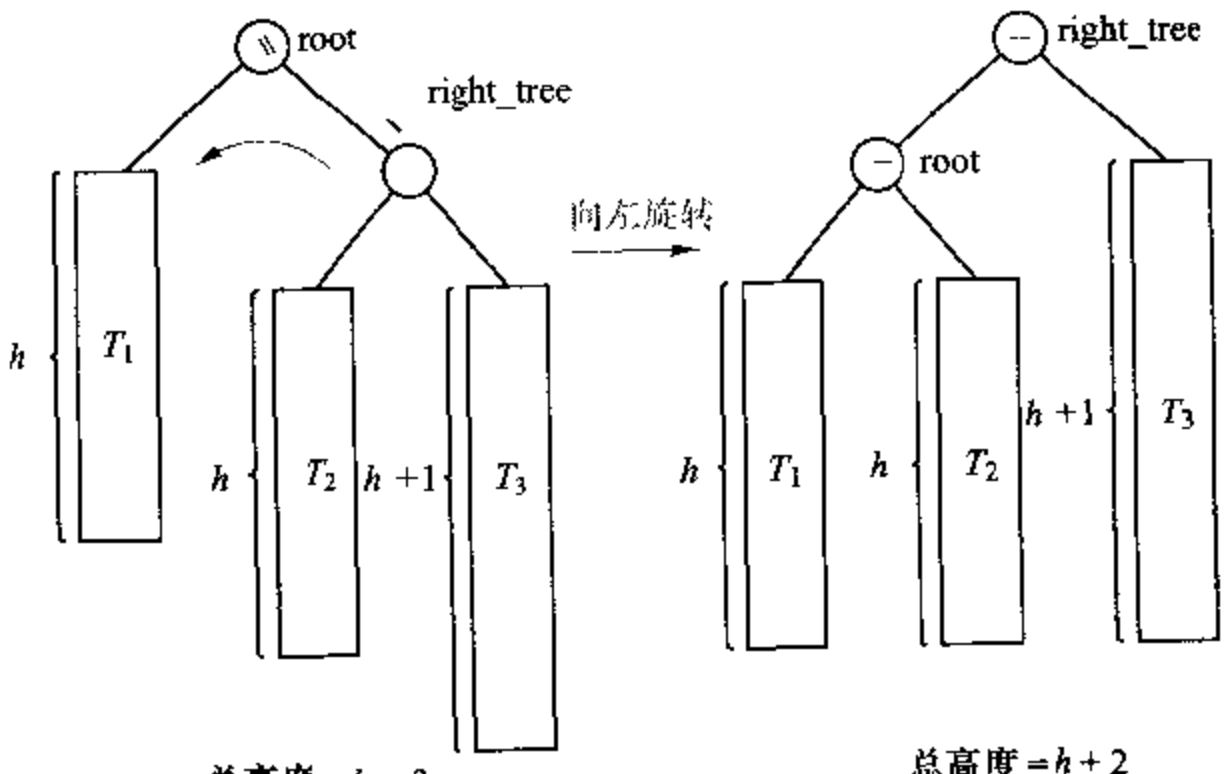


图 10.19　第一种情况：通过左旋转恢复平衡

```
}
```

4. 情况 2：左高

* * * 双旋转

第二种情况是 right_tree 的平衡因子属于左高，这种情况稍微有点复杂。为了找到新的根，必须移动两层到结点 sub_tree，它是 right_tree 的左子树的根。这个过程显示在图 10.20 中，因为能在两步内实现转换，因此又称为双旋转：首先右旋转以 right_tree 为根的左子树（以使 sub_tree 变成它的根），然后左旋转 root 指向的树（将 sub_tree 往上移，使它成为新的根）。

第二种情况中，root 和 right_tree 的新平衡因子依赖于 sub_tree 以前的平衡因子。（sub_tree 的新平衡因子将始终是 equal_height。）图 10.20 将 sub_tree 的子树显示成具有相同的高度，但有可能 sub_tree 是左高或者右高。结果得到的平衡因子是：

原 sub_tree	新 root	新 right_tree	新 sub_tree
—	—	—	—
/	—	\	—
\	/	—	—

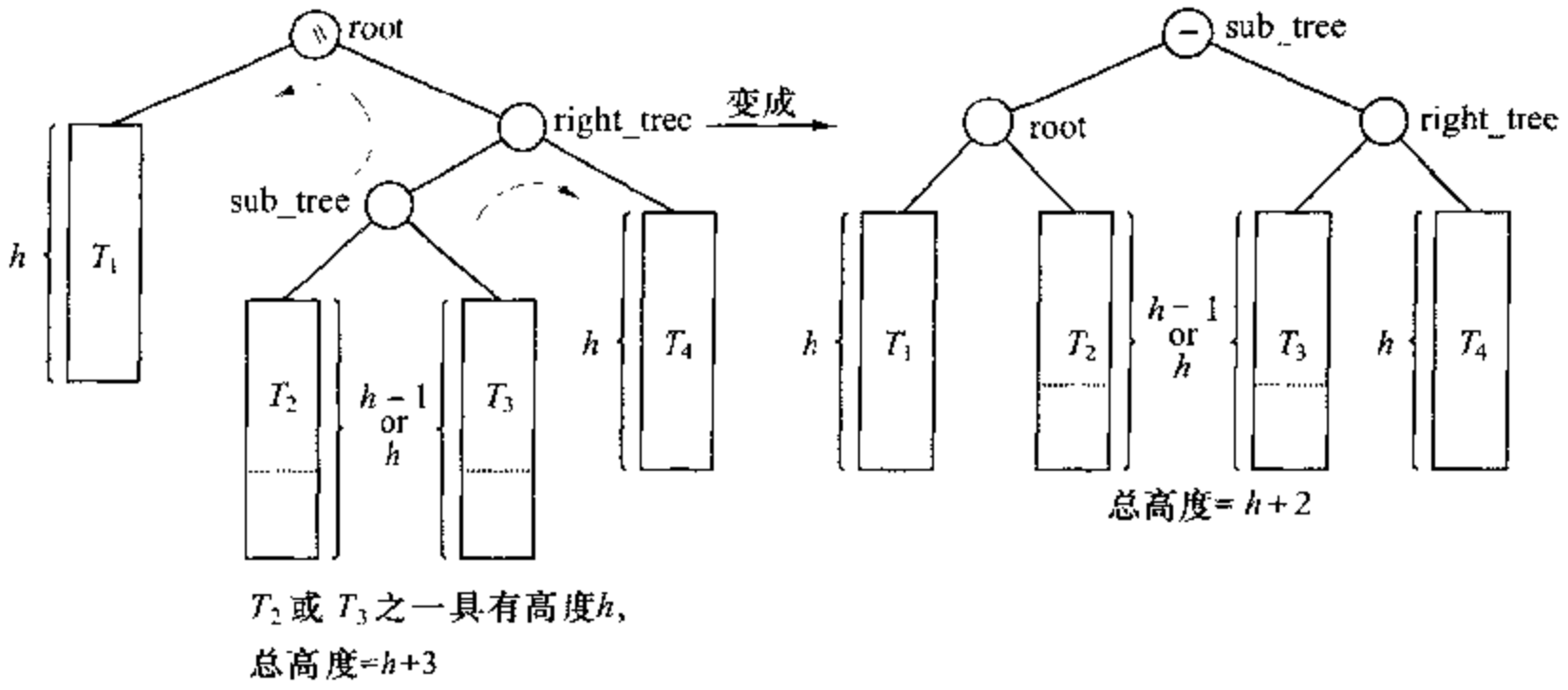


图 10.20 第二种情况:通过双旋转恢复平衡

5. 情况 3:等高

最后,看来我们必须考虑第三种情况,即 right_tree 的两棵子树具有同样的高度,但这种情况实际上决不会发生。为了明了为何是这样,我们还记得刚刚将一个新结点插入到以 right_tree 为根的子树中,而且这棵子树的高度现在比 root 的左子树多 2。新结点进入 right_tree 的左子树或右子树,因此它的插入只增加了 right_tree 的一个子树的高度。如果这些子树在插入后具有同样的高度,以 right_tree 为根的整棵子树的高度并没有因插入而改变,这与我们已知的相反。

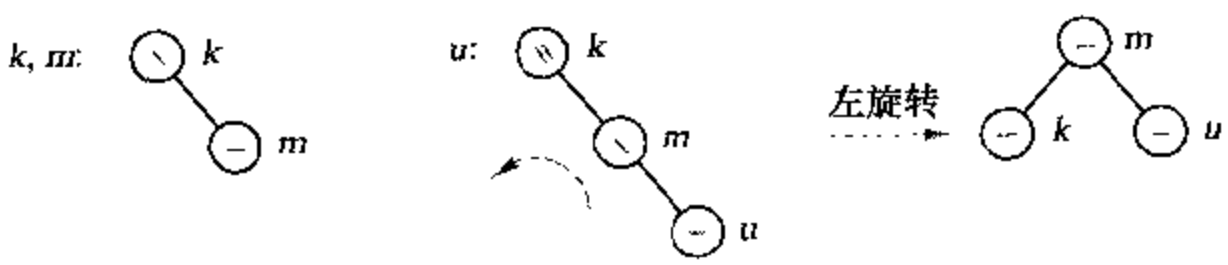
6. 用于平衡的C++函数

现在将这些转换并入C++函数就很简单了,函数 rotate_right 和 left_balance 的形式明显分别类似于 rotate_left 和 right_balance 的形式,它们将留作练习。

```
template <class Record>
void AVL_tree<Record>::right_balance(Binary_node<Record>*&sub_root)
/*Pre: sub_root points to a subtree of an AVL_tree that is doubly unbalanced on the
       right.
  Post: The AVL properties have been restored to the subtree.
  Uses: Methods of struct AVL_node;
       functions rotate_right and rotate_left.*/
{
  Binary_node<Record>*&right_tree=sub_root->right;
  switch (right_tree->get_balance( )){
  case right_higher:          //  single rotation left
    sub_root->set_balance(equal_height);
    right_tree->set_balance(equal_height);
    rotate_left(sub_root);
    break;
  case equal_height:          //  impossible case
    cout<<"WARNING: program error detected in right_balance"<<endl;
  case left_higher:           //  double rotation left
```

```
    Binary_node<Record>*sub_tree=right_tree->left;
    switch (sub_tree->get_balance( )){
    case equal_height:
      sub_root->set_balance(equal_height);
      right_tree->set_balance(equal_height);
      break;
    case left_higher:
      sub_root->set_balance(equal_height);
      right_tree->set_balance(right_higher);
      break;
    case right_higher:
      sub_root->set_balance(left_higher);
      right_tree->set_balance(equal_height);
      break;
    }
    sub_tree->set_balance(equal_height);
    rotate_right(right_tree);
    rotate_left(sub_root);
    break;
  }
}
```

需要单旋转和双旋转的插入例子如图 10.21 所示。



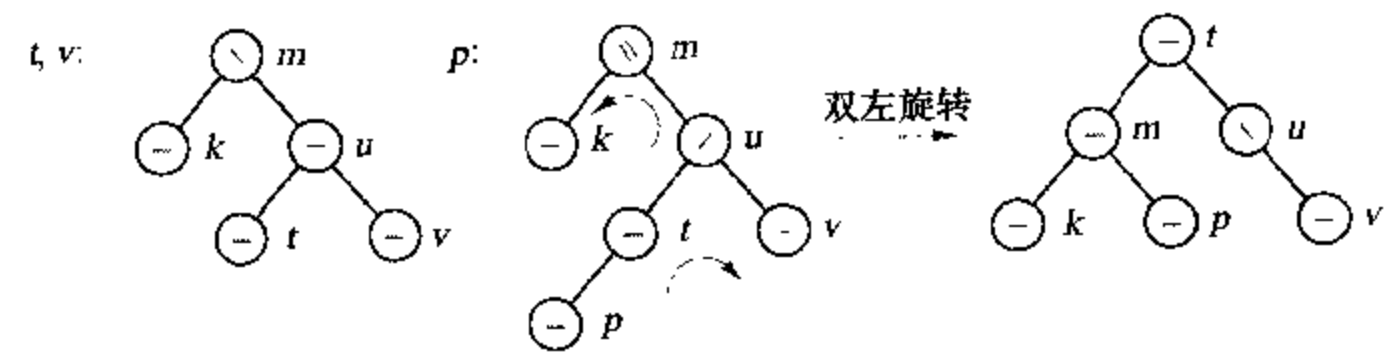


图 10.21　要求旋转的 AVL 插入

7. 算法的性能

* * * 计算旋转次数

函数 avl_insert 递归调用本身去插入一个新结点的次数可能与树的高度一样多。乍看起来，似乎这些调用中的每一个都可能导致合适子树的单旋转或者双旋转，但实际上，至多只会进行一次(单或双旋转)。为了解这一点，我们回忆一下，只有在函数 right_balance 和 left_balance 中才做旋转，而且仅当子树高度增加时才能调用这些函数。然而当这些函数返回后，旋转已消除了增加的高度，因此，对剩余的(外部)递归调用，高度等于没有增加，因此没有进一步做旋转或修改平衡因子。

AVL 树的大多数插入不会导致旋转。甚至在需要旋转时，这些旋转也通常发生在刚插入的树叶附近。即使插入到 AVL 树的算法很复杂，还是有理由期望它的运行时间与插入到同样高度的普通查找树的时间略有不同。稍后会看到，我们能期望 AVL 树的高度比随机查找树的高度小得多，因此 AVL 树中的插入和检索都会比随机查找树高效得多。

10.4.3 结点的删除

将结点 x 从 AVL 树中删除要求同样的基本思想，包括用于插入的单旋转和双旋转。我们将仅给出非正式的方法框架步骤，编写完整算法的任务作为程序设计项目。

** * * 方法*

1. 将问题减少到要删除的结点 x 至多只有一个孩子的情况。假设 x 有两个孩子，则首先取 x 的左孩子，然后将它尽可能远地右移，找到中序遍历下 x 的直接前趋 y（直接后继将一样）。找到 y 的方法保证结点 y 没有右孩子。将 y（或 y 的副本）放入树中 x 占据的位置（具有和 x 同样的双亲、左右孩子以及平衡因子）。现在将 y 从其先前位置上删除，如下继续，在下列步骤中，用 y 代替 x。
2. 从树中删除结点 x。由于我们知道（通过第一步）x 有至多一个孩子，通过将 x 的双亲链到 x 的单个孩子删除 x（若 x 无孩子，则 x 的双亲链设为 NULL）。以前以 x 为根的子树高度减少 1，并且我们现在必须利用从 x 回到树根的路径上的所有结点去追踪高度发生这种改变所带来的影响。我们使用了一个布尔型的变量 shorter 去表明子树的高度是否已缩短。在每个结点采用的操作依赖于 shorter 的值和此结点的平衡因子，有时还依赖于此结点的孩子的平衡因子。
3. 布尔变量 shorter 的初始值为真。倘若 shorter 保持真，将对从 x 的双亲到树根的路径上的所有结点 p 执行下列步骤。当 shorter 变为假时，则不需要进一步的改变，并且算法终止。
4. 情况 1：现在的结点 p 有相等的平衡因子。当 p 的左或右子树已被缩短，p 的平衡因子相应地改变，且 shorter 变为假。
5. 情况 2：p 的平衡因子不相等，且较高的子树被缩短。改变 p 的平衡因子直到相等，且保持 shorter 为真。
6. 情况 3：p 的平衡因子不相等，且较短的子树被缩短。对 AVL 树的高度要求现在在 p 处遭到破坏，因此我们运用如下的旋转恢复平衡。让 q 成为 p 的较高子树的根（未被缩短的那棵子树），而根据 q 的平衡因子又分成 3 种情况。
7. 情况 3a：q 的平衡因子相等。单个旋转（改变 p 和 q 的平衡因子）即恢复平衡，且 shorter 变为假。
8. 情况 3b：q 的平衡因子与 p 相等。运用一个单旋转，设置 p 和 q 的平衡因子相等，并保持 shorter 为真。
9. 情况 3c：p 和 q 的平衡因子相反。运用一个双旋转（首先围绕 q，然后是围绕 p），设置新根的平衡因子为相等并且设置其他结点合适的平衡因子，并保持 shorter 为真。

在情况 3a，3b 和 3c 中，旋转的方向依赖于是左子树还是右子树被缩短。图 10.22 说明了其中一些可能性，图 10.23 给出了删除结点的一个例子。

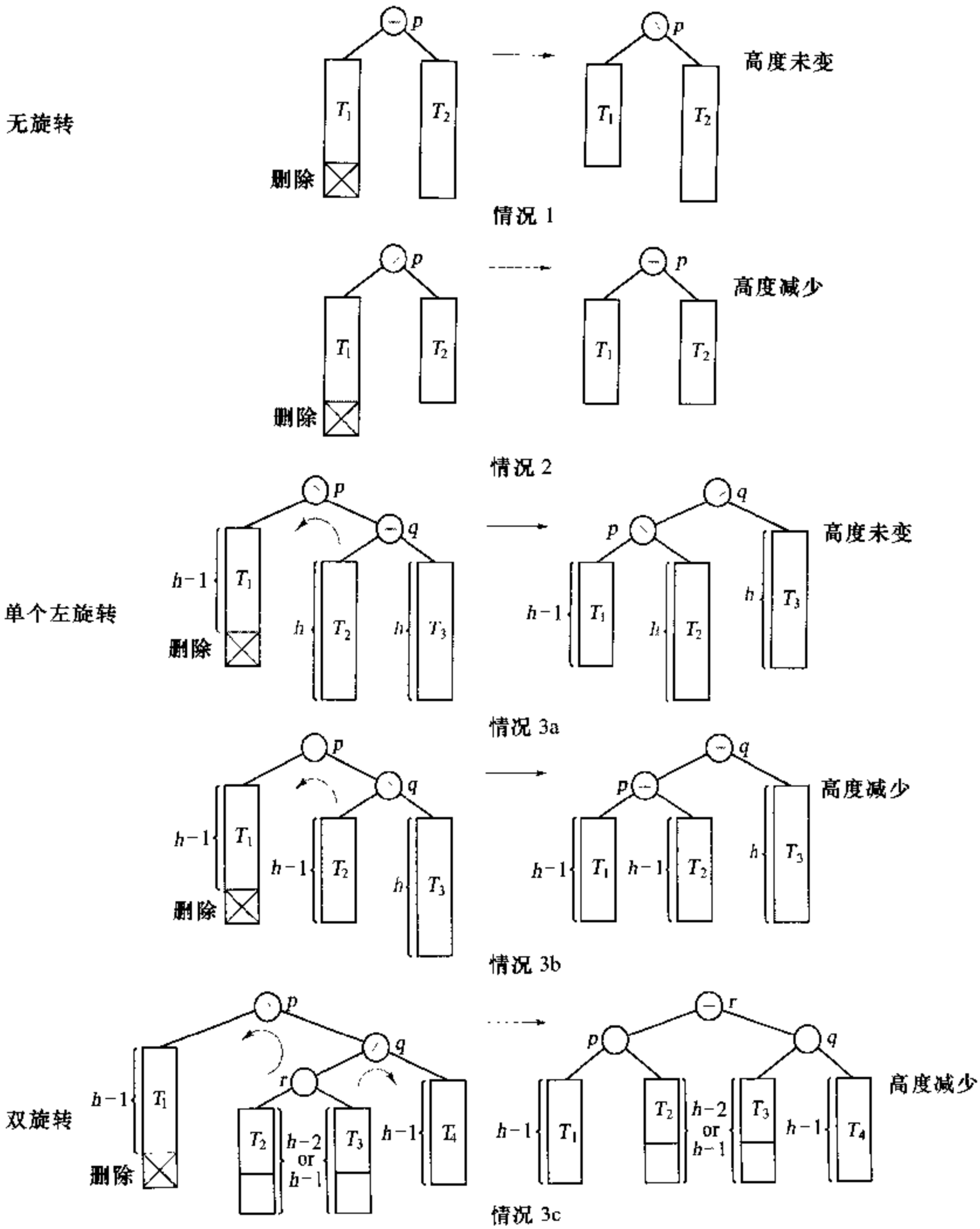


图 10.22　简单情况：从 AVL 树删除

图 10.23 从 AVL 树删除的例子

10.4.4　AVL 树的高度

＊＊＊最坏情况分析

找到平均 AVL 树的高度非常困难，因此用本节的算法去确定平均做多少步也非常困难。然而，找到最坏情况下发生的事情则要简单得多，而且这些结果表明 AVL 树的最坏性能实质上几乎不会差于随机树的性能。经验证据表明 AVL 树的平均性能比随机树好得多，几乎和完全平衡树所能获得的性能一样好。

＊＊＊最坏情况分析

要确定 n 个结点的 AVL 树可能具有的最大高度，我们能反过来询问高度为 h 的 AVL 树可能具有的最少结点数是多少。如果 F_h 是这样的一棵树，它的根的左子树和右子树是 F_l 和 F_r，则 F_l 和 F_r 中必然有一个高度为 $h-1$，比方说 F_l，另一个的高度或者为 $h-1$ 或者为 $h-2$。由于在高度为 h 的 AVL 树中，F_h 具有最少的结点数，随之可以得出：在高度为 $h-1$（即 F_l 具有形式 F_{h-1}）的 AVL 树中，F_l 必然具有最少的结点数，且 F_r 必须具有最少结点数的 $h-2$ 高度（因此，F_r 具有形式 F_{h-2}）。

＊＊＊斐波纳契树

通过前述规则建立的树，对于 AVL 树来说是尽可能稀疏的，称为斐波纳契树。图 10.24 显示了最初的几棵树。

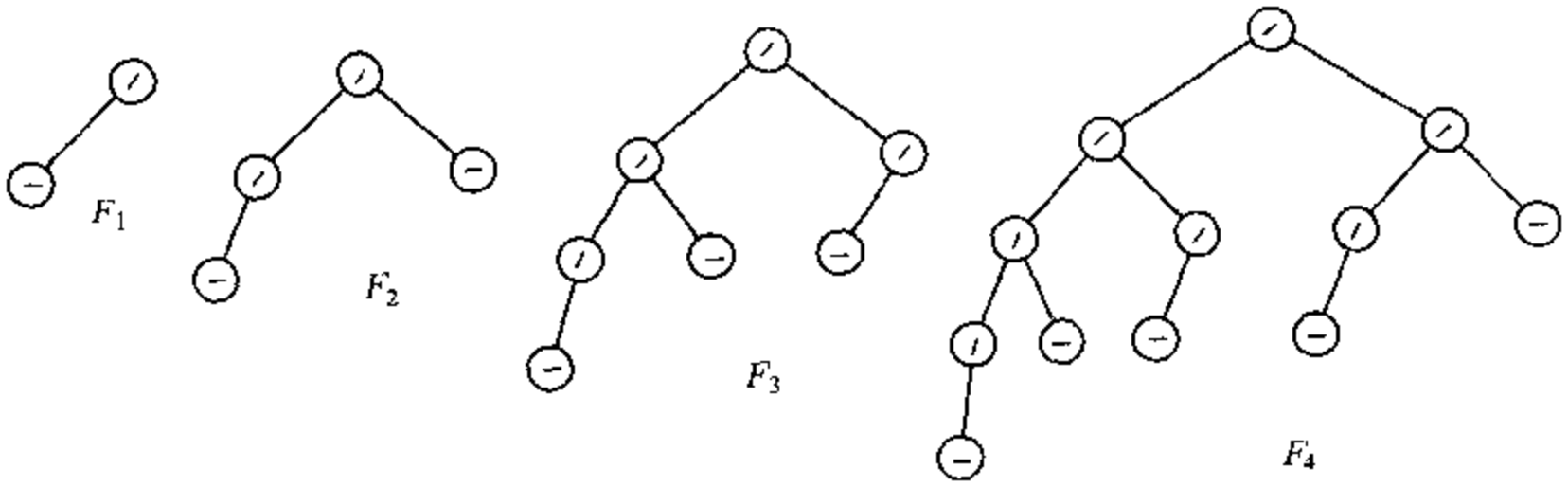


图 10.24　斐波纳契树

＊＊＊计算斐波纳契树的结点数

如果设|T|是树 T 的结点数，则有（计算根及子树）递归关系：

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1$$

这儿 $|F_0|=1$，$|F_1|=2$。通过在两边都加 1，我们看到数 $|F_h|+1$ 满足斐波纳契数的定义（见 A.4 节），只是下标改成 3。通过 A.4 节中斐波纳契数的求值，我们看到：

$$|F_h|+1 \approx \frac{1}{\sqrt{5}}\left\lfloor\frac{1+\sqrt{5}}{2}\right\rfloor^{h+2}$$

＊＊＊斐波纳契树的高度

接下来，我们通过在两边取对数和放弃除最高项以外的项来求解 h。近似的结果是：

$$h \approx 1.44\lg|F_h|$$

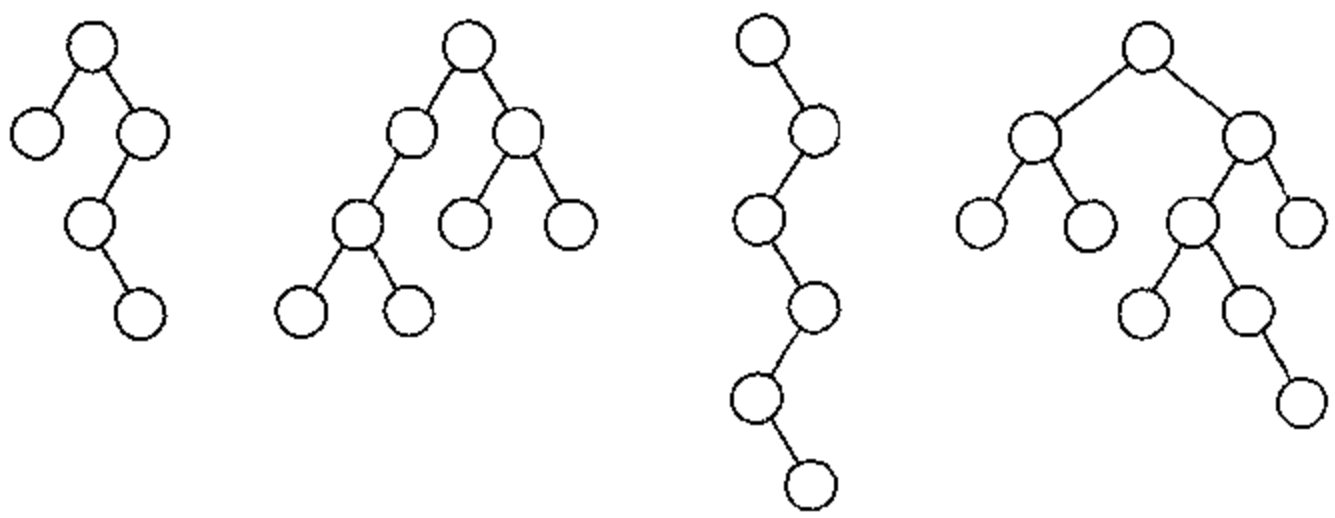
* * * 最坏情况的边界

* * * 平均情况

这意味着具有 n 个结点的最稀疏可能的 AVL 树高度近似为 1.44 lgn。一个具有 n 个结点的完全平衡的二叉树的高度约为 lgn，退化树的高度和 n 一样。因此操作 AVL 树的算法保证花费的时间只是比最优时间多 44%。实际上，AVL 树比这个效果更好。即使对 AVL 树的最坏情况斐波纳契树，也能表明其平均查找时间仅比最优情况多 4%。大多数 AVL 树不是象斐波纳契树那样稀疏，因此有理由期望一般 AVL 树的平均查找次数确实非常接近最优次数。实际上，经验研究表明，当 n 比较大时，平均比较次数大约为 lgn +0.25。

练习 10.4

E1. 确定下列哪一棵二叉查找树是 AVL 树。对于那些不是 AVL 树的，找出违背要求的所有结点。



(a) (b) (c) (d)

E2. 在下列每个序列中，以所示的次序插入键，用它们建立 AVL 树。

(1)A,Z,B,Y,C,X

(2)A,B,C,D.E,F

(3)M,T,E,A,Z,G,P

(4)A,Z,B,Y,C,X,D,W,E,V,F

(5)A,B,C,D,E,F,G,H,I,J,K,L

(6)A,V,L,T,R,E,I,S,O,K

E3. 以 LIFO 次序(最后一个插入的键最先删掉)从 AVL 树中删除练习 E2 插入的每个键。

E4. 以 FIFO 次序(第一个插入的键最先删掉)从 AVL 树中删除练习 E2 插入的每个键。

E5. 从下图(见下页)的 AVL 树开始，删除下列每个键。独立地进行每次删除，每次都从原始树开始。

(1)k (4)a (6)m

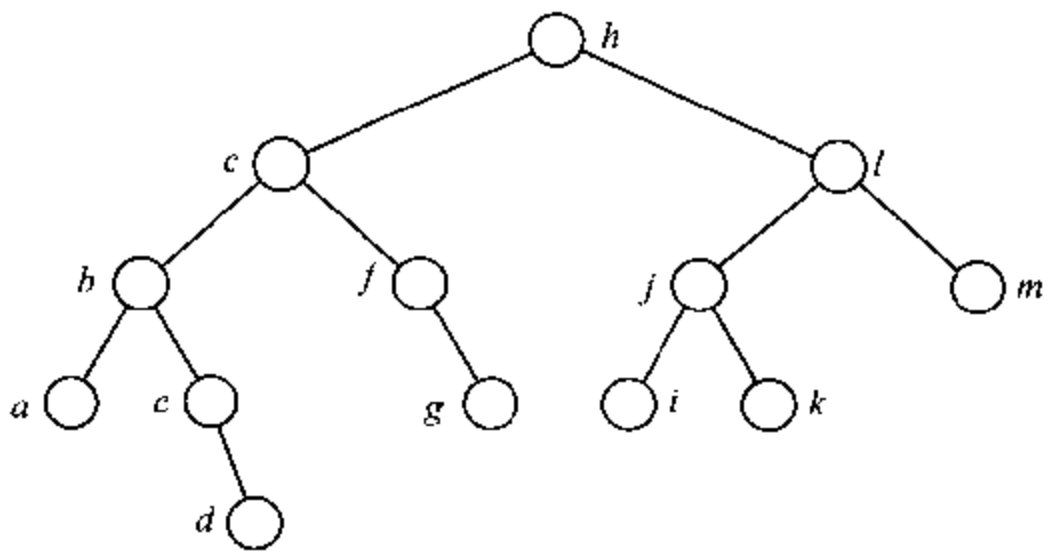
(2)c (5)g (7)h

(3)j

E6. 编写方法，通过仅跟踪到树叶的一条路径，而不查看树中的所有结点，返回 AVL 树的高度。

E7. 编写函数，它返回一个指针，指向最接近非空 AVL 树根的最左树叶。

E8. 证明从 AVL 树中删除一个键所做的旋转次数(单或双旋转)不能超出此树高度的一半。



程序设计项目 10.4

P1. 编写C++方法，从 AVL 树删除一个结点，遵循课文中概述的步骤。

P2. 将 AVL 树类替换到 10.2 节项目 P2 为二叉查找树设计的菜单驱动程序中，因此得到 AVL 树的一个演示程序。

P3. 将 AVL 树类替换到 10.2 节项目 P5 的信息检索项目中。用各种组合的输入文本文件，比较 AVL 树的性能和普通二叉查找树的性能。

10.5　伸展树：自我调节的数据结构

10.5.1　引言

＊＊＊医院记录

考虑医院里维护病人记录的问题。在院病人的记录是极其活跃的，不断地被医生和护士参考和更新。当病人离开医院时，他的记录变得不太活跃，但此病人的医生或其他人仍不时需要它。如果后来病人重新入院，则他的记录突然又变得极其活跃。而且，因为这次入院可能是紧急的，即使不活跃的记录也应该快速可用，而不仅仅是作为备份文档保存，使得对它的访问很慢。

＊＊＊访问时间

如果我们使用普通的二叉查找树，或甚至是 AVL 树去保存医疗记录，则新入院的病人将进入到远离根的树叶位置，因此访问将很慢。而我们希望将新近插入或经常访问的记录保存在接近根的地方，不活动的记录放在较远的地方，接近或就在树叶上。但我们不能关闭医院的系统去重建期望形状的树，即使是关闭系统一个小时。因此需要把树建成一种自我调节的数据结构，可以自动地改变树的形状，使频繁使用的记录更靠近根结点，而允许不活跃的记录慢慢地移到叶结点。

＊＊＊自我调节的树

伸展树是二叉查找树，它以一种非凡的方法通过自我调节达到我们的目标：在我们每次访问树结点时，不管是插入或是检索，我们对树执行根本的手术，一路向上抬高新访问的结

点，使它成为被修改树的树根。其他的结点在必要时要让路给新结点，为它腾出空位。经常访问的结点将频繁地被抬升而成为根结点，而且它们将不会离顶部位置太远。另一方面，不活跃的结点将慢慢地被推得离根越来越远。

*** 平摊分析

伸展树可能变得高度不平衡，因此，对树结点的一次访问可能代价很大。然而在这一节的后面，我们将证明，经过一个较长序列的访问后，伸展树的代价根本不高，而且能保证所需的操作甚至不比 AVL 树更多。我们所使用的分析工具称为平摊算法分析，因为像保险计算一样，经过一个较长序列的操作，将极少的高代价事件平均分摊在许多代价不高的事件中，从而得到优越的性能。

我们通过使用在 AVL 树上使用的那些类似形式的旋转对伸展树实施基本的手术，但现在对于树中的每次插入或检索要使用许多旋转。事实上，旋转是沿着从根到被访问的目标结点的路径完成。我们现在要准确地讨论这些旋转是如何进行的。

10.5.2 伸展步骤

*** zig 和 zag

如图 10.19 所示，当在二叉查找树中执行单个旋转时，树中的一些结点抬高了，一些结点降低了。在一次左旋转中，双亲结点往下移一层，而它的右孩子上移一层。如图 10.20 所示，一次双旋转，由两次单旋转组成，一个结点向上移了两层，而所有其他的结点上移或下移了至多一层。从刚被访问的目标结点开始，沿着向上到根的路径逐步发展，我们在每一步能作单旋转，因此将目标结点一路上移到根。这个方法将实现把目标结点变成根结点的目标，但它表明，平摊到许多访问上的树的性能可能并不好。

取而代之，伸展的关键思想是每一步将目标结点上移两层。首先是一些简单的术语：考虑从根向下到达被访问结点的路径。每次沿这条路径向下左移，我们说 zig，而每次右移，我们说 zag。两步左移（往下）称为 zig-zig，两步右移称为 zag-zag，左移然后右移称为 zig-zag，右移然后左移称为 zag-zig。这 4 种情况是沿着路径向下移动两步的所有可能。然而如果路的长度是奇数，则在末尾需要多一步，或者是 zig（左）移动或是 zag（右）移动。

图 10.25 显示了对每个 zig-zig，zig-zag 和 zig 移动在伸展中进行的旋转，其他 3 种情况 zag-zag，zag-zig 和 zag 则是它们的镜像。

图 10.25 中的 zig-zag 情况等同于图 10.20 中的一次 AVL 双旋转，zig 等同于单旋转（图 10.19）。然而，zig-zig 情况不同于通过两次利用单旋转提高目标结点所得，如图 10.26 所示。

*** 移两层

为了避免图 10.26 中所示的错误，我们总是考虑每次（除了末尾仅剩的单个 zig 或 zag 步骤）将目标升高两层。同时注意只是从目标到根的路径上的结点其相对位置改变了，如图 10.25 中显示的仅被浅灰色虚曲线围住的那些结点。路径外的子树（在图 10.25 中显示为 $T1$，$T2$，$T3$，$T4$）都没有改变其形状，这些子树在此路径上所挂之处正是它们可能继续保持所有键的查找树次序的惟一位置。

我们通过学习图 10.27 中显示的一个例子来巩固头脑中的这些思想。

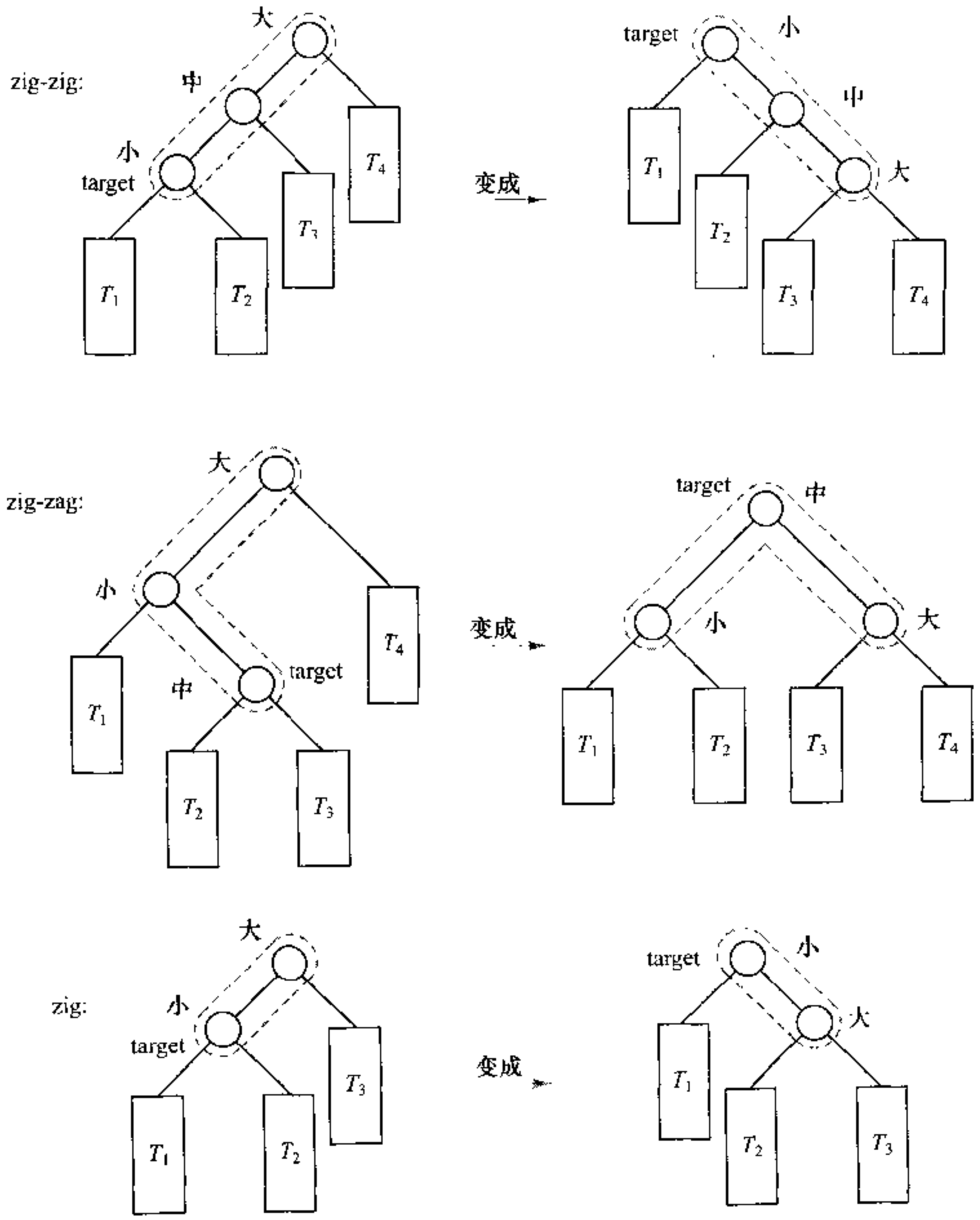


图 10.25　伸展旋转

＊＊＊例子

我们从顶层左子树开始，在 c 处伸展。从根到 c 的路径经过了 h,f,b,e,d,c。从 e 到 d 到 c 是 zig-zig(左-左，由虚线椭圆显示)，因此，我们在树的这部分执行 zig-zig 旋转，得到图 10.27 中第二棵树。注意树的剩余部分没有改变形状，而且被修改的树悬挂在它最初占据的同一位置上。

下一步，从 f 到 b 到 c 的路径(显示在第二棵树中的虚线内)现在是一个 zig-zag 移动，而且结果得到的 zig-zag 旋转生成第三棵树。因此子树 d,e(离开路径)没有改变形状，但移到

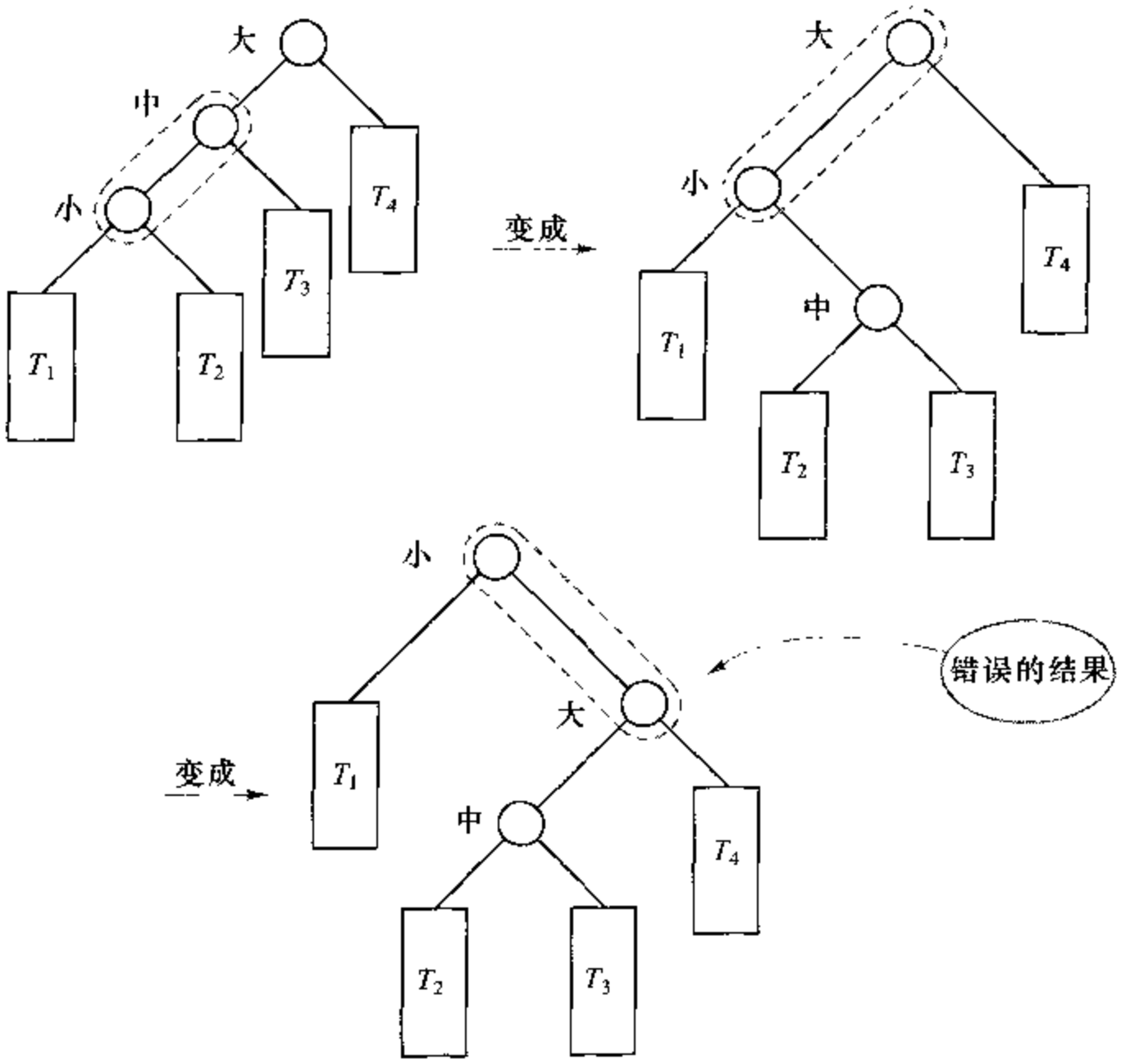


图 10.26 用单旋转错误地代替 zig-zig

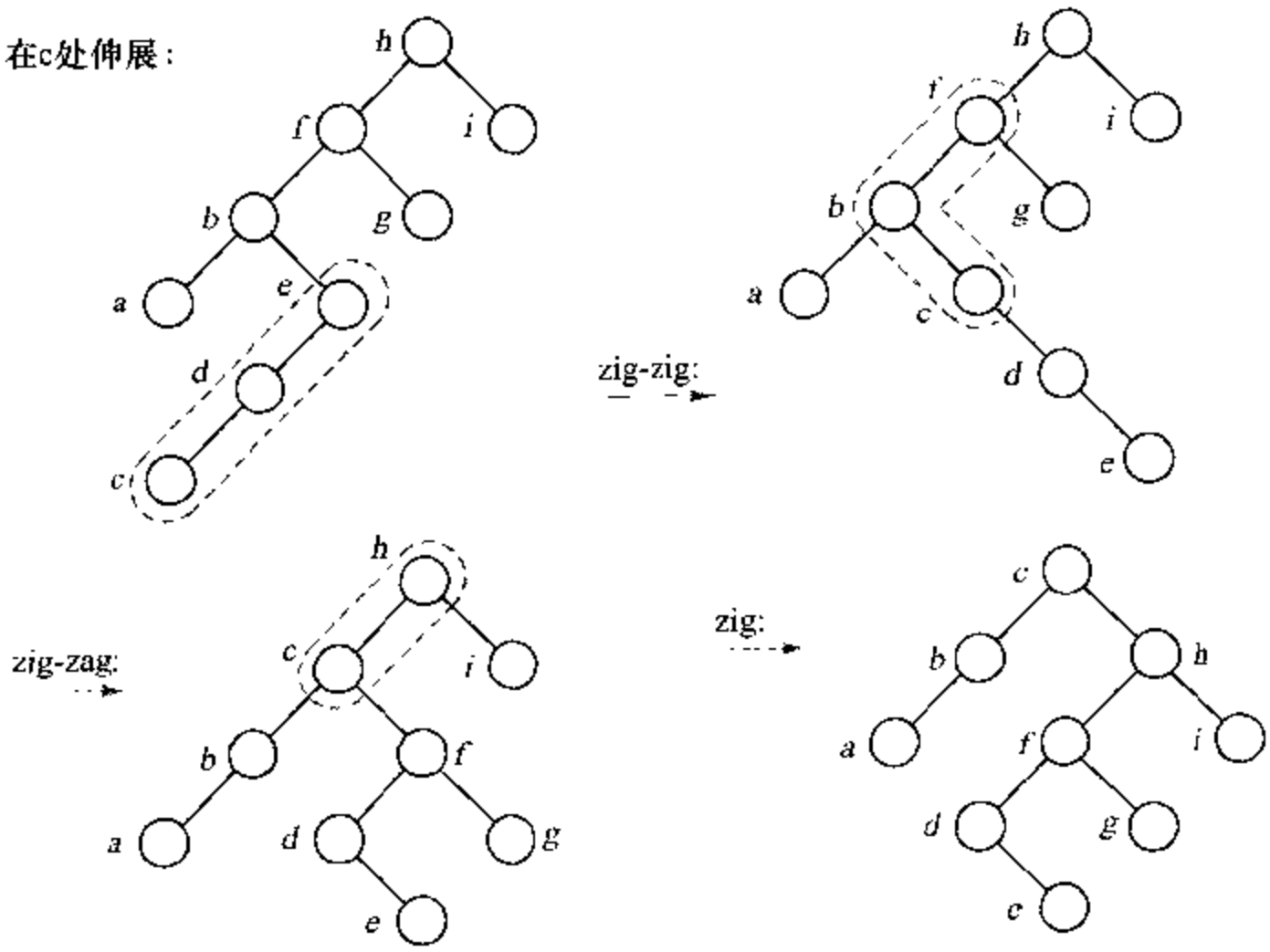


图 10.27 伸展的例子

了一个新位置，如图 10.25 中显示的 T_3。

在这第三棵树中，c 离根仅有一步，它在根的左子树上，因此一个 zig 旋转生成图 10.27 中最终的树。这儿，以 f 为根的子树没有改变形状，但改变了位置。

* * * 自底向上的伸展

在这个例子中，我们执行了自底向上的伸展，即从目标点开始，每次两步地沿路径上移到根。在手工完成的例子中，这是一个自然的方法，因为在自顶向下的查找之后，一个结点将期望回转并自底向上伸展回到树的顶部。因此，如果必要，在过程的最后做一个单 zig 或 zag 移动，就出现在树顶。自底向上的伸展实质上是一个“两遍”的方法，首先是向树下查找，然后向上伸展目标直到根。实际上，如果没有找到目标则插入此结点，那么自底向上的伸展可能需要“三遍”设计：一遍用于查找；一遍用于插入；而一遍用于伸展。

* * * 自顶向下的伸展

然而在编写计算机算法时，在查找目标结点的同时自顶向下伸展证明更加简单有效。当我们发现目标时，立即将它移到树根，或者如果查找未成功，则创建含有目标的新根。在自顶向下的伸展中，单一的 zig 或 zag 移动发生在伸展过程的底部。

因此，如果运行我们为例子中的树所设计的伸展函数，当目标被移动偶数层时，可以得到与手工做自底向上伸展的同样的结果，但当移动奇数层时，结果将不会相同。

10.5.3　算法设计

我们只设计一个伸展函数，它能同时用于检索和插入。给定一个目标键，此函数将在树中查找这个键，并在移动过程中伸展。如果它找到了这个键，则检索它；如果没有找到，则此函数将它作为新结点插入。在这两种情况中，含有目标键的结点都将作为树根结束。

我们将伸展树作为 Search_tree 类的派生类实现，因此，除了特殊的伸展方法以外，所有通常的 Search_tree 方法都能用于伸展树。类说明如下：

```
template <class Record>
class Splay_tree:public Search_tree<Record>{
public:
  Error_code splay(const Record &target);
                                  //  Add auxiliary function prototypes here.
private:
};
```

伸展树将不跟踪高度，不使用任何像 AVL 树那样的平衡因子。

1. 三路树划分

* * * 三路树拆分

自顶向下的伸展使用了与图 10.25 中说明的同样的移动(zig-zig，zig-zag 和其他)，但是当伸展进行时，根必须为空，从而使得结束时目标结点能被移到根中或直接插入到根中。因此，当伸展进行时，树临时分成独立的子树，在目标已成为根之后，这些子树被重新连接。我们如下使用 3 棵子树：

- 中央子树(central subtree)包含的结点，如果目标存在，则一定在这些结点中。
- 较小键子树(smaller-key subtree)子树包含那些键严格小于目标的结点；事实上，较

小键子树中的每个键都比中央子树中的每个键小。

- 较大键子树(larger-key subtree)包含键严格大于目标的结点;实际上,较大键子树中的每个键大于中央子树中的每个键。

* * * 三路不变量

这些条件在整个伸展过程中都将保持真,因此我们称它们为三路不变量。

最初,中央子树是整棵树,较小键和较大键子树是空的,因此,三路不变量最初时正确。随着查找的进行,结点从中央子树剥离,并加入到其他两棵子树中的一棵。当查找结束时,如果存在目标结点,则中央子树的根将是目标结点,如果未找到目标结点,则中央子树将为空。在任一种情况里,所有的组件和作为根的目标将最终连在一起,见图 10.28。



图 10.28 在伸展中拆开的三路树

2. 基本操作:link_right

在查找的每个阶段,我们将目标和中央子树树根的键进行比较。假设目标较小,则查找将移到左边,于是我们能取根及其右子树并将它们连接到较大键子树中,而将中央子树减小到根以前的左子树,我们称这个过程为 link_right,因为它将右边的结点链到较大键子树中。其操作如图 10.29 所示。

* * * zig 与 link_right 的比较

注意 link_right 与 zig 移动之间的类似性:在两种情况中,将左孩子结点向上移去代替它的双亲结点,而双亲结点向下移到右子树中。实际上,除了从先前的左孩子向下到先前的双亲的链被删除外,link_right 恰是 zig 移动,取而代之的是双亲(和它的右子树)移到了较大键子树。

* * * 移到双亲

在较大键子树中,这个双亲(先前中央子树的根)将附在何处?三路不变量告诉我们中央子树中的每个键都在较大键子树中的每个键之前,因此这个双亲(和它的右子树)必须附在较大键子树中最左结点(键序列中的第一个)的左边。图 10.29 中显示了这个过程。特别要注意,在 link_right 执行后,三路不变量继续为真。

3. 设计伸展操作

link_right 操作实现一个 zig 变换,并且它的对称类似物 link_left 将执行 zag 变换,但大多数时间,伸展每次需要两层移动。奇怪的是,所有剩余的伸展操作仅能使用 link_right 和

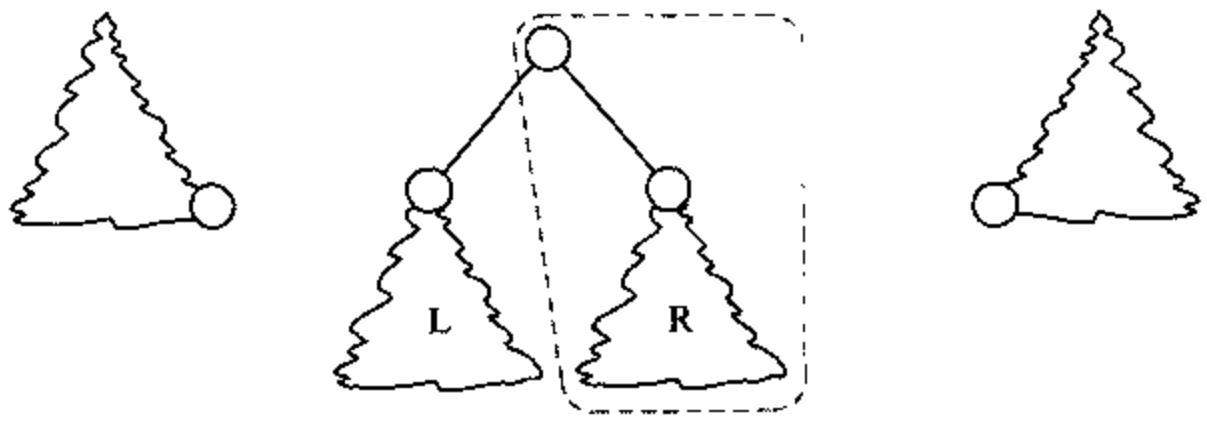


图 10.29　link_right 操作:zig 移动

link_left 以及普通的(单个)左右旋转去执行。

首先考虑 zig-zig 情况,如图 10.30 中说明。在这个情况中,目标键不仅小于中央子树的根(称为 large),而且小于它的左孩子(称为 middle),所以根及其左孩子和它们的右子树都应该被移到较大键子树中。为此,我们首先围绕它的根右旋转中央子树,然后执行 link_right。

注意图 10.30 中相对位置的改变与 zig-zig 移动一样:从 large 到 middle 到 small 左移的链变成了右移的链;small 沿树上移,而且虽然从 small 到 middle 的右链丢失了,这条链还有一个到 large 的右链。同时注意,在最后的安排中,三路不变量继续成立。

在图 10.31 说明的 zig-zig 情况中,目标键来自根键(large)及其左孩子(small)之间的某个位置,因此 large 及其右子树能被移到较大键子树,而且 small 和它的左子树能被移到较小键子树。通过首先做 link_right 然后做 link_left,可以实现这些步骤。再次注意,在此过程完成后,三路不变量继续成立。

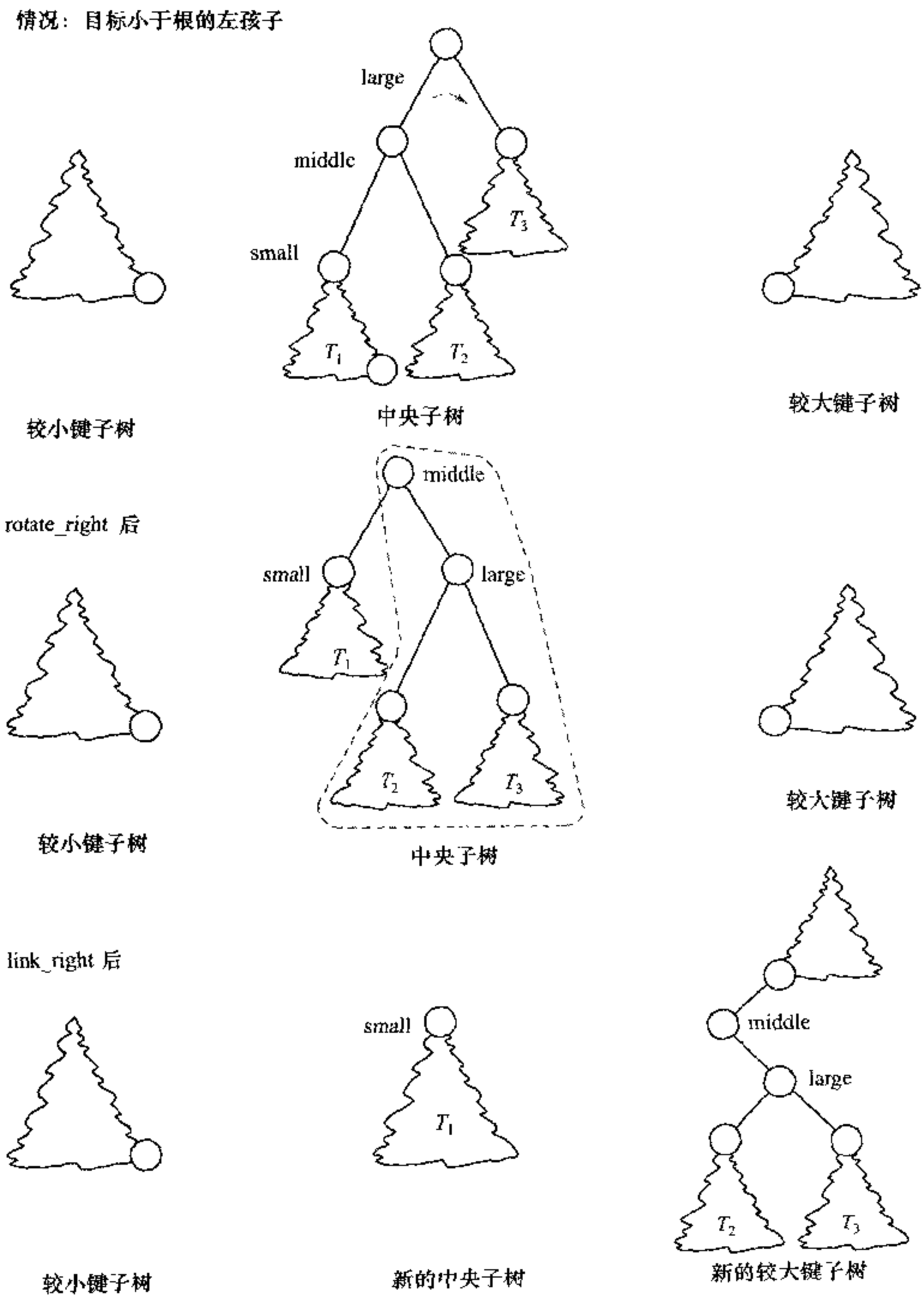
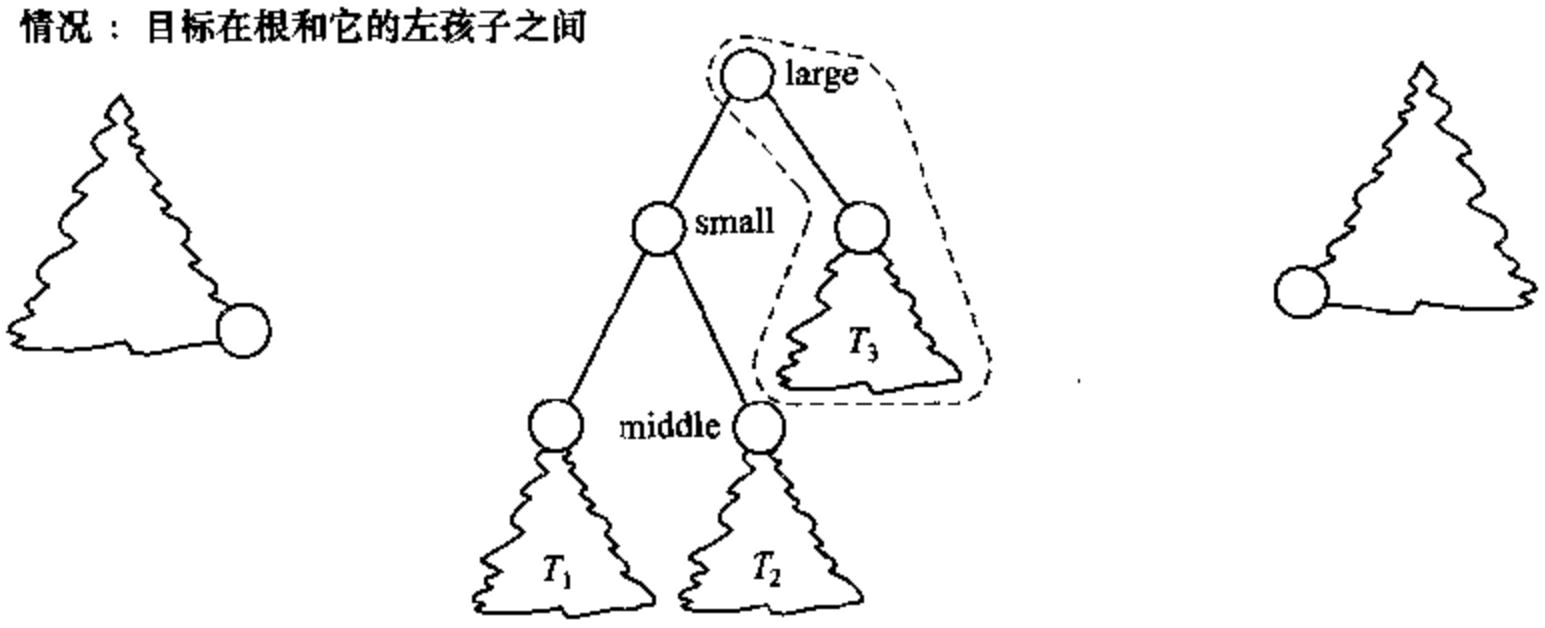


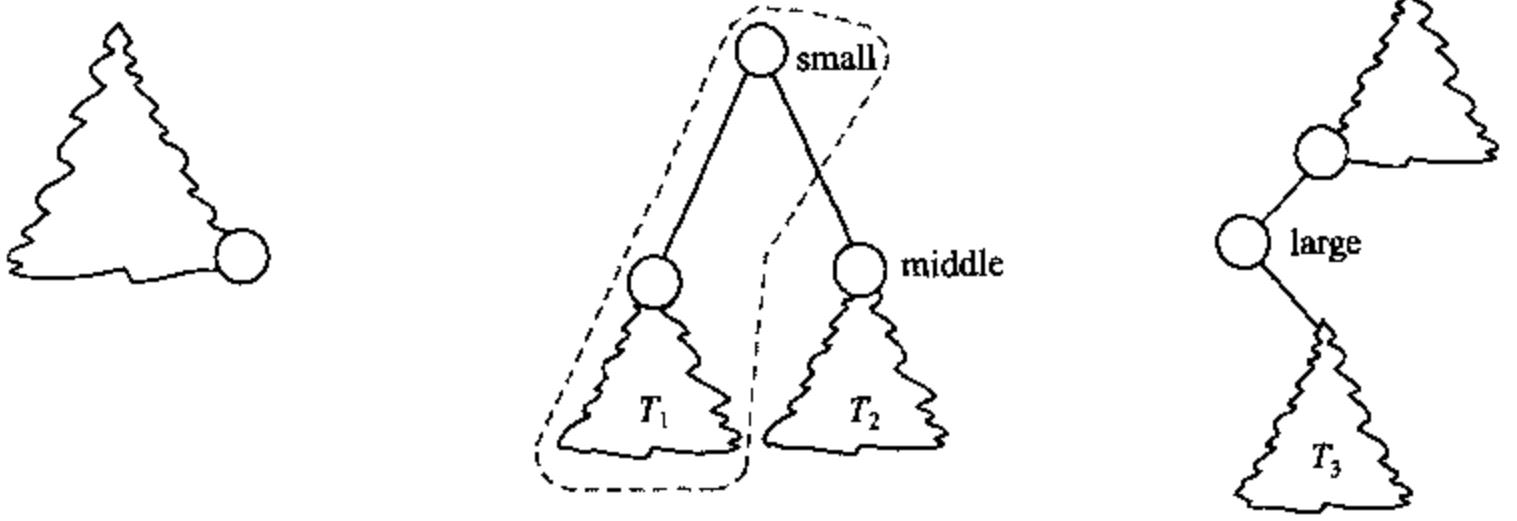
图 10.30 作为 rotate_right, link_right 执行的 zig-zig

情况：目标在根和它的左孩子之间



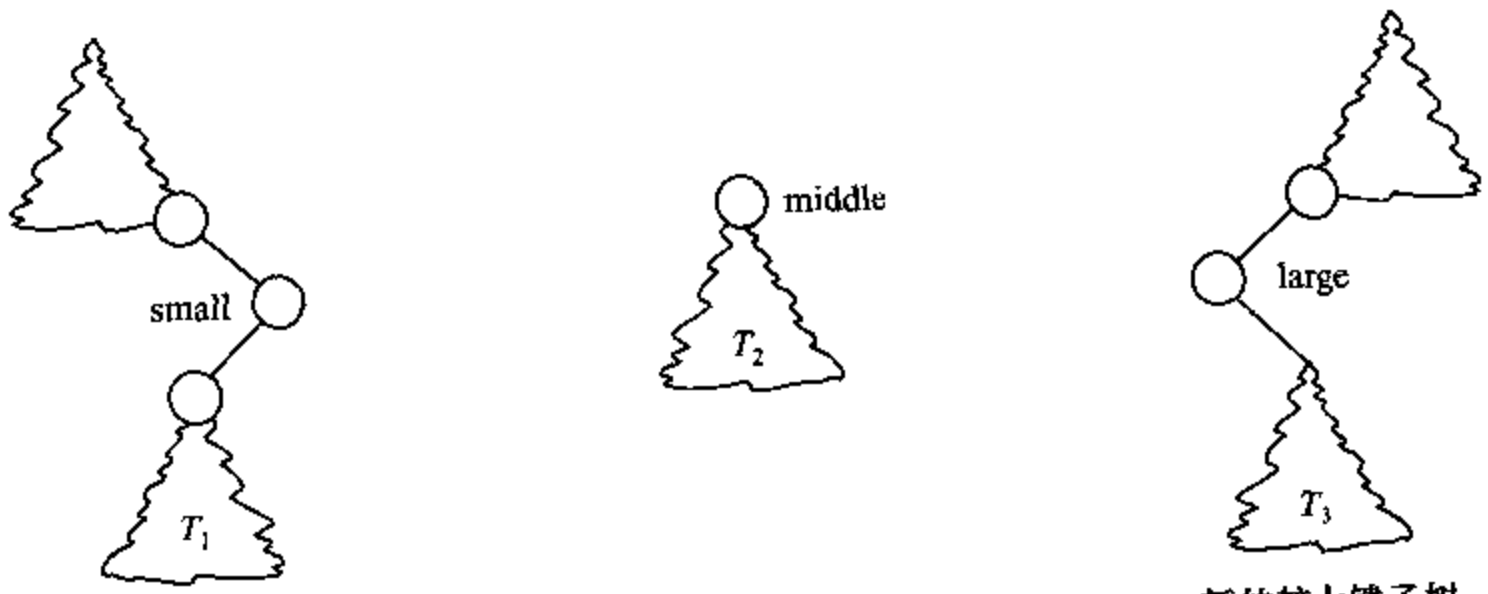
较小键子树　　中央子树　　较大键子树

link_right 后：



较小键子树　　中央子树　　新的较大键子树

link_left 后：



新的较小键子树　　新的中央子树　　新的较大键子树

图 10.31　作为 link_right, link_left 执行的 zig-zag

4. 程序设计约定

* * * 变量

我们设计的算法使用了 5 个指针变量去跟踪 3 棵子树中需要的位置，其中前 4 个指针如下：

- current 给出了尚未查找的结点的中央子树的根。
- child 根据需要引用 current 的左孩子或右孩子。
- last_small 给出较大键子树中的最大(也即，最右)结点，这是附加的结点必须依附的位置。
- first_large 给出了较大键子树中的最小(也即，最左)结点，这是附加的结点必须依附的位置。

* * * 问题：空子树

* * * 问题：丢失的根

我们现在开始编码函数，从 link_right 开始，我们立即发现必须解决的两个问题。首先是空子树的问题。由于较大键子树初始为空，因此将 first_large 初始化为 NULL 是合理的。我们然后在每次执行 link_right 时需要检查这种情况，否则我们将试图跟随一个 NULL 指针。其次，我们必须有某种方法去找到较小键和较大键子树的根，从而在伸展结束后，我们能将它们链回到一起成为一棵树。迄今，我们还没有建立指向这些根的指针，而对伸展本身，我们并不需要它们。

解决这些问题的一个方法是在我们的函数内引进条件语句去检查各种特殊情况。而我们介绍一种新的程序设计策略，它在许多其他应用中相当有用，同时它将免除我们检查空指针的需要和(以后我们会见到)免除跟踪子树的根的需要。

* * * 哑结点

这个程序设计策略是使用一个附加的结点，称为哑结点(dummy)，它不包含键或其他数据，而且方便时将只是代替空树，也就是说，当一个空链被解除引用时可使用它。因此我们的第 5 个即最后一个指针变量是：

- dummy 指向一个哑结点，这个结点在伸展函数开始时创建并在它结束时删除。

为了指示较小键和较大键子树初始为空，我们将 last_small 和 first_large 都初始化指向 dummy。这样，last_small 和 first_large 将总是指向某个实际的 Binary_node，因而 link_right 和 link_left 将不会试图去解除引用空指针。

5. 辅助函数

有了所有这些决定，对 link_right 和 link_left 进行编码就是一件简单的事情。

```
template <class Record>
void Splay_tree<Record> :: link_right (Binary_node<Record>*&current,
                                       Binary_node<Record>*&first_large)
/*Pre: The pointer first_large points to an actual Binary_node (in particular, it is
       not NULL). The three-way invariant holds.
  Post: The node referenced by current (with its right subtree) is linked to the right of
       the node referenced by first_large. The pointer first_large is reset to
       current. The three-way invariant continues to hold. */
```

```
{
  first_large->right=current;
  first_large=current;
  current=current->left;
}

template <class Record>
void Splay_tree<Record>::link_left (Binary_node<Record>*&current,
                                    Binary_node<Record>*&last_small)
/*Pre: The pointer last_small points to an actual Binary_node (in particular, it is not
       NULL ). The three-way invariant holds.
  Post: The node referenced by current (with its left subtree) is linked to the left of
        the node referenced by last_small. The pointer last_small is reset to current.
        The three-way invariant continues to hold .*/
{
  last_small->left=current;
  last_small=current;
  current=current->right;
}
```

旋转函数也易于编码，它们不使用哑结点，而且它们不会引起三路划分中的任何改变。

```
template <class Record>
void Splay_tree<Record>::rotate_right(Binary_node<Record>*&current)
/*Pre: current points to the root node of a subtree of a Binary_tree. This subtree has
       a nonempty left subtree.
  Post: current is reset to point to its former left child, and the form ercurrent node
        is the right child of the new current node */
{
  Binary_node<Record>*left_tree=current->left;
  current->left=left_tree->right;
  left_tree->right=current;
  current=left_tree;
}

template <class Record>
void Splay_tree<Record>::rotate_left(Binary_node<Record>*&current)
/*Pre: current points to the root node of a subtree of a Binary_tree. This subtree has
       a nonempty right subtree.
  Post: current is reset to point to its former right child, and the former current node
        is the left child of the new current node */
{
  Binary_node<Record>*right_tree=current->right;
  current->right=right_tree->left;
  right_tree->left=current;
  current=right_tree;
}
```

6. 完成任务

* * * 终止

当查找完成时，中央子树的根结点指向目标结点或为空。如果找到目标 target，则它必成为整棵子树的根，但在这之前，可以知道它的左子树和右子树现在分别属于较小键子树和较大键子树，因此应将它们移到那儿。如果查找因不成功而终止，则有 current==NULL，则

必须创建一个包含 target 的新根。

* * * 重新组装树

最后，新根的左、右子树现在成为较小键子树和较大键子树。现在我们必须返回到尚未解决的第二个问题：由于我们让指针各自仅指向这些子树的最右和最左的结点，则如何找到这些子树的根？

* * * 解决方案：不要遗失根

为了回答这个问题，回忆在查找开始时所发生的事情。最初，last_small 和 first_large 这两个指针都设置为指向 dummy 结点。当一个结点(及子树)加入较大键子树时，它们是通过改变 first_large->left 而加入较大键子树的左边。由于 first_large 初始时是 dummy，则在查找结束时我们能够找出插入较大键子树的第一个结点，因此它的根简化为 dummy->left。类似地，dummy->right 指向较小键子树的根。因此 dummy 结点提供了指向较小键子树和较大键子树的根的指针，否则就会丢失这些根。但指针不是存储在可能期望保留的位置。

这些步骤在图 10.32 中说明。

7. 伸展：最终方法

有了已完成的所有这些预备工作，最后我们已能编写函数去在二叉查找树中真正检索和插入一个结点，同时能伸展这棵树以使该目标结点进入树根。

```
template <class Record>
Error_code Splay_tree<Record>::splay(const Record &target)
/*Post: If a node of the splay tree has a key matching that of target, it has been moved by
      splay operations to be the root of the tree, and a code of entry_found is
      returned. Otherwise, a new node containing a copy of target is inserted as the
      root of the tree, and a code of entry_inserted is returned.*/
{
  Binary_node<Record>*dummy=new Binary_node<Record>;
  Binary_node<Record>*current=root,
                     *child,
                     *last_small=dummy,
                     *first_large=dummy;
//  Search for target while splaying the tree.
  while (current!=NULL && current->data!=target)
    if (target<current->data){
      child=current->left;
      if (child==NULL || target==child->data)          //  zig move
        link_right(current, first_large);
      else if (target<child->data){                    //  zig-zig move
        rotate_right(current);
        link_right(current,first_large);
      }
      else {                //  zig zag move
        link_right(current, first_large);
        link_left(current, last_small);
      }
    }
    else {                  //  case:target>current->data
      child=current->right;
      if (child==NULL || target==child->data)
```

```
        link_left(current,last_small);                    //  zag move
      else if (target>child->data){                       //  zag-zag move
        rotate_left(current);
        link_left(current, last_small);
      }
    else {                          //  zag-zig move
      link_left(current,last_small);
      link_right(current,first_large);
    }
  }
//  Move root to the current node, which is croated if necessary.
  Error_code result;
  if (current==NULL){                                 //  Search unsuccessful: make a new root.
    current=new Binary_node<Record>(target);
    result=entry_inserted;
    last_small->right=first_large->left=NULL;
  }
  else {                                              //  successful search
    result=entry found;
    last_small->right=current->left;                  //  Move remaining central nodes.
    first_large->left=current->right;
  }
  root=current;                                       //  Define the new root.
  root->right=dummy->left;                            //  root of larger-key subtree
  root->left=dummy->right;                            //  root of smaller-key subtree
  delete dummy;
  return result;
}
```

* * * 结论

现在已考虑了所有的事情，在指针的操作和资源的经济使用上，这个算法确实非常精细和复杂。

10.5.4　平摊算法分析：介绍

我们现在希望确定在较长序列的操作上进行伸展的性能，但在完成这个任务之前，我们用简单的例子介绍算法的平摊分析。

1. 引言

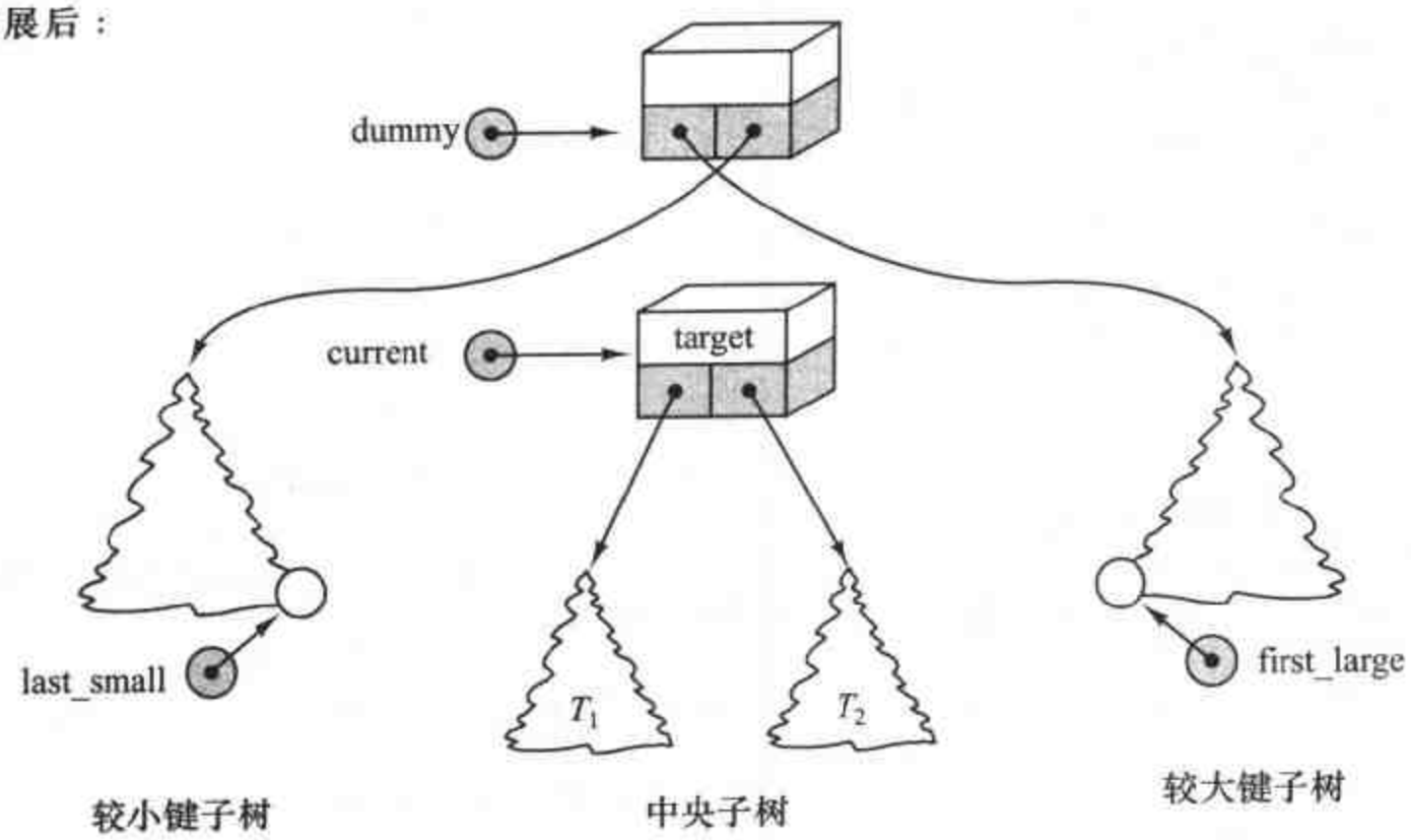
* * * 定义

过去，我们已考虑了两种进行算法分析的方法：最坏情况分析和平均情况分析。在这两种分析方法中，我们采用单一事件或单一情形并试图确定算法为处理它而做了多少工作。平摊分析与这两种方法的不同之处在于：它考虑一个长序列的事件，而不是一个孤立的单一事件。平摊分析给出了长序列事件的代价的最坏情况估计。

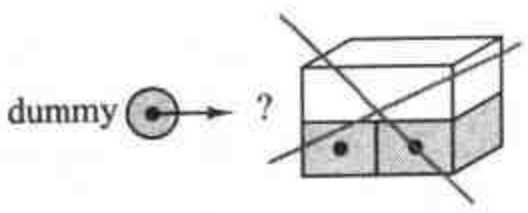
* * * 多重弹出

很可能序列中的一个事件会影响后来事件的代价。一个任务可能执行困难且代价高昂，但它能让数据结构处于某个状态，使得随后的任务变得更加简单。例如，考虑一个栈，任

伸展后：



重新连接成一棵树：



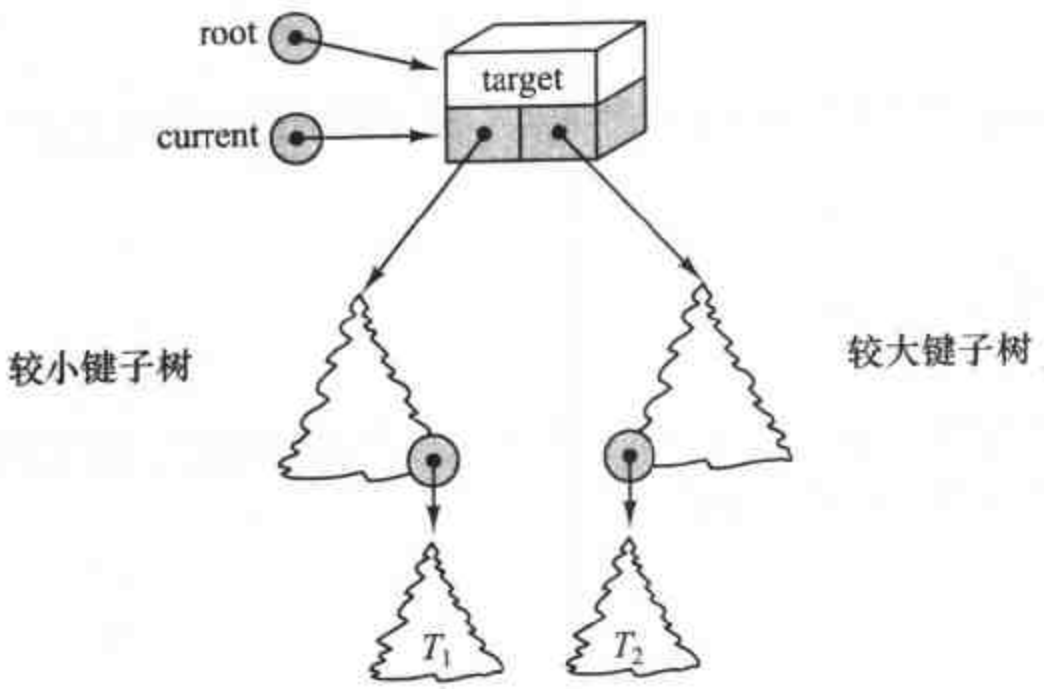


图 10.32 重新连接子树

意数目的元素都可以一起放入这个栈中，任意数目的元素也能一起从这个栈中弹出。如果栈中有 n 个元素，因为所有的元素可能一起弹出，则多重弹出操作的最坏代价明显是 n。然而，如果几乎所有的元素（以一种昂贵的弹出操作）被弹出，则因为栈中剩余元素很少，后续的弹出操作不会昂贵。我们设弹出 0 元素的代价为 0，则如果我们从栈中 n 个元素开始，做一系列的 n 多重弹出，则即使最坏情况的代价是 n，每个弹出平摊的最坏情况代价仅为 1，其

原因在于 n 个多重弹出操作总共只能从栈中删除 n 个元素，因此它们的总代价不会超过 n。

* * * 平摊

在金融领域，平摊意味着在一段时间上分布大量的花费，比如使用抵押以多月偿还来平摊房子(有利息)的花费，会计在收益上平摊所用的大量资金，保险精算师将高风险事件平摊到一般性人口上。

2. 平均与平摊的比较

* * * 独立事件与相关事件的比较

平摊分析与平均情况分析不同，因为前者考虑的是一系列相关的情形，而后者考虑所有可能的独立情形。以排序方法为例，我们在所有可能的事件上作平均情况分析。而谈论在同一个表的某一行上进行两次排序是没有意义的，因此平摊分析通常不用于排序。

* * * 排序

然而，我们能设计一个平摊分析起作用的例子。考虑一个起始有序的表，在对表做了一些使用后，要将一个新元素插入到表中的随机位置。在进一步使用后，表又再次被排序。后来，另一个元素又随机插入，以此类推。我们应该使用什么排序方法？如果依赖于平均情况分析，可能我们选择快速排序。如果我们更愿意做最坏情况分析，则可能选择归并排序或堆排序以保证 $O(n \log n)$ 的性能。然而平摊分析将导致我们采用插入排序：一旦这个表已有序且要随机插入一个新元素，插入排序能以 $O(n)$ 的性能将它移到合适的位置。因为表接近有序，快速排序(具有最好的平均情况性能)可能给出最坏的实际性能，因为支点的某些选择将导致它接近其最坏情况。

3. 树遍历

作为另一个例子，考虑二叉树的中序遍历，在这里我们将访问一个顶点的代价估计为从最近访问的顶点到达该顶点所遍历的树枝数。图 10.33 显示了三棵二叉树，中序遍历显示成一条浅灰色路径，访问每个顶点的代价也用浅灰色显示。

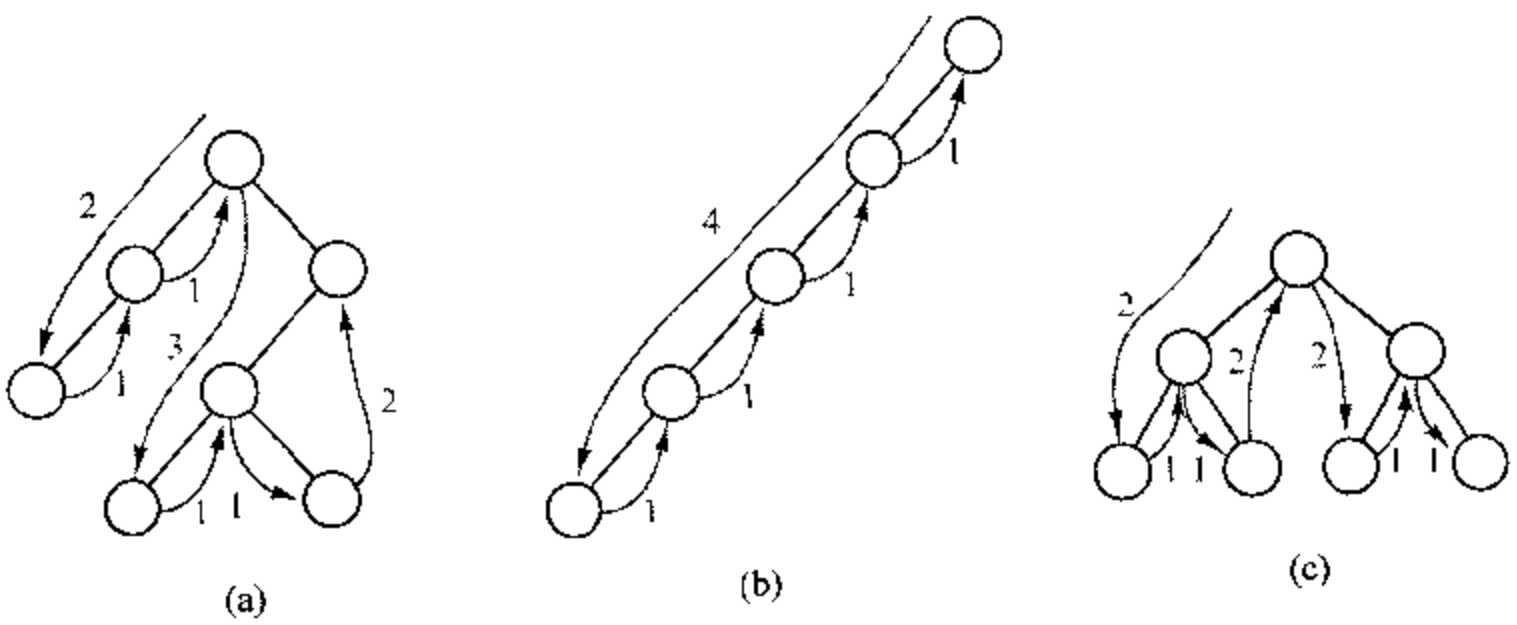


图 10.33　中序遍历的代价

当我们从一个顶点到它的一个孩子或其双亲，访问一个顶点的最佳情况代价是 1。对具有 n 个顶点的树，最坏情况代价是 $n-1$，这棵树显示的是往左的一条长链，到达第一个(最左)顶点的分支是 $n-1$。(见图 10.33(b)部分。)然而在这条链中，所有剩下的顶点遍历时从

每个顶点到其双亲仅仅一步就可到达。在规模为 n 的完全平衡二叉树中，一些顶点需要 $\lg n$ 步，其他的仅需要 1 步。(见图 10.33(c)部分。)

然而，如果我们在整棵二叉树的遍历上平摊代价，则从每个顶点到下一个顶点的代价小于 2。为了解这一点，首先注意到每棵具有 n 个顶点的二叉树正好有 $n-1$ 个分支，因为除根以外的每个顶点刚好只有一个分支到达它。树的完全遍历在每个分支上要走两次，一次往下，一次往上。(这儿我们已包括在访问最后、最右顶点之后沿着路径往上走到根。)因此在整个遍历中，总共的步数是 $2(n-1)$，从一个顶点到下一个顶点的平摊步数是 $2(n-1)/n<2$。

4. 存款余额：使代价平衡

假设你正在做家庭预算。如果你受雇了，则(你希望)你的收入通常每个月将相当稳定。然而，你的开销却未必如此。有些月份要将大量的钱花在保险、学费或较大的购物支出上，而其他月份则可能没有特殊的花费。为了保持你的银行存款有偿还能力，你需要在开销小的月份存入足够的钱，从而能在需要时偿付大额的账单。在有大额账单的月份开始时，你有大量的银行存款，而在付过账之后，你的银行存款会变少，但你还是很幸运，因为现在你只欠下较少的钱。

*** 存款函数

我们希望将这种思想用于算法分析。为此，我们设计了一个称为存款余额的函数，它的运转就像一个预算良好的家庭能够做到收支平衡。存款函数将按如下方式选择：当下一个操作昂贵，则函数值将很大，而当下一个操作能很快做完时，它变得较小。我们将存款函数看作有助于负担一些昂贵操作的代价，而对不昂贵的操作，我们比实际代价多留出一些，使用超额去增大存款函数以便将来使用。

为了使这种思想更精确，假设我们在某个数据结构上有 m 个操作的序列，对 $1\leqslant i\leqslant m$，让 t_i 表示操作 i 的实际代价。设我们的存款函数的值是 $c_0, c_1, \cdots\cdots, c_m$，这里 c_0 是第一个操作前的存款余额，对 $1\leqslant i\leqslant m$，c_i 是操作 i 后的存款余额。于是我们给出基本定义：

定义 对 $i=1,2,\cdots\cdots,m$，每个操作的平摊代价 a_i 定义为：

$$a_i=t_i+c_i-c_{i-1}$$

这里的 t_i, c_i 如刚才定义。

这个等式说明平摊代价是实际代价加上操作执行期间已改变的存款余额。

记住我们的存款余额只是一个记账工具：我们自由地设计想要的函数，但其中有一些会比另外的更好。我们的目标是帮助预算，因此我们的目标是：

*** 目标

选择存款余额函数 c_i 使得平摊代价尽可能地接近相等，而不管实际代价 t_i 如何变动。

我们现在希望使用平摊代价去帮助计算 m 个操作序列总共的实际代价。基本的定义重新安排为 $t_i=a_i+c_{i-1}-c_i$，于是总共的实际代价则为：

$$\sum_{i=1}^{m} t_i=\sum_{i=1}^{m}(a_i+c_{i-1}-c_i)=(\sum_{i=1}^{m} a_i)+c_0-c_m$$

除了第一个值和最后一个值，所有的存款余额都互相抵消了，因此它们不参与最后的计算。

为了将来的引用，我们将这个事实作为引理重新表述：

引理 10.5　在某个数据结构上的一系列的 m 个操作总共的实际代价及总共的平摊代价之间的关系表述为：

$$\sum_{i=1}^{m} t_i = (\sum_{i=1}^{m} a_i) + c_0 - c_m$$

我们的目标是以 a_i 接近相等的方式去选择存款余额函数，于是这个等式的右端就易于计算，因此一系列的 m 个操作总共的实际代价相同。

＊＊＊套叠和

像这样的和式，它的项交替出现加号和减号，因而当相加时，它们会互相抵消，这样的和称为套叠和(telescoping sum)。因为它可能会让你记起一种玩具式的(或便携式的)套筒，它由一些短管组成，这些管子互相滑动，可以扩展形成一个长的套筒管。

5. 增加二进制整数

我们现在通过学习另一个简单的例子来巩固这些思想，然后我们就可以运用这些思想去证明关于伸展树的基本且令人惊奇的定理了。

＊＊＊例子

我们考虑的这个例子是一个算法，它连续地对一个二进制(基为 2)整数加 1。我们从右边开始，如果当前位是 1，我们就将它变为 0，然后左移，当我们到达极左侧或遇到一个 0 位时就停止，在停止前我们将这个 0 位变为 1。执行这个算法的代价是改变的位数(二进制数字)。这个算法在一个 4 位整数上运用 16 次后的结果显示在图 10.34 的前 3 列中。

第 i 步	整数	t_i	c_i	a_i
	0000		0	
1	0001	1	1	2
2	0010	2	1	2
3	0011	1	2	2
4	0100	3	1	2
5	0101	1	2	2
6	0110	2	2	2
7	0111	1	3	2
8	1000	4	1	2
9	1001	1	2	2
10	1010	2	2	2
11	1011	1	3	2
12	1100	3	2	2
13	1101	1	3	2
14	1110	2	3	2
15	1111	1	4	2
16	0000	4	0	0

图 10.34　增加二进制整数的代价和平摊代价

对这个例子中的存款余额，我们取此二进制整数中 1 的个数。显然，必须改变的数字个数恰比这个整数最右部分中 1 的个数多 1，因此 1 越多，要改变的数字也越多。有了这个选择，使用基本的公式，我们就能计算每一步的平摊代价。此结果显示在图 10.34 的最后一列，除了最后一个为 0 以外，其他的每一步都显示为 2。因此我们得出结论，能用改变 2 个数字的平摊代价去增加一个 4 位数的二进制整数，即使实际代价是从 1 到 4 变化。

10.5.5 伸展的平摊分析

完成所有这些预备介绍后，我们现在能用平摊算法分析技术来确定伸展树算法在一个长系列的检索和插入上做了多少工作。

* * * 实际复杂性的度量

作为实际复杂性的度量，我们将取伸展前目标结点拥有的树的深度，当然这是结点将在树中上移的位置数。算法的所有操作——键的比较和旋转——在锁定的步骤里与这个深度相关。例如，函数所做的主循环的迭代趟数是这个深度的一半。

* * * 概念

首先，我们介绍一些简单的概念。设 T 是我们执行伸展插入或检索的二叉查找树，T_i 表示伸展过程的 i 步后转换得到的 T，其中 $T_0=T$。如果 x 是 T_i 中的任意结点，则我们用 $T_i(x)$ 表示根为 x 的子树，且 $|T_i(x)|$ 表示这棵子树中的结点数。

我们假设在结点 x 处伸展，而且考虑自底向上的伸展，因此 x 在 T 中的某处开始，但在 m 步伸展后，作为 T 的根结束。

* * * 秩函数的定义

对伸展过程的每一步 i 和 T 中的每一个顶点 x，我们定义 x 在第 i 步的秩为：$r_i(x)=\lg|T_i(x)|$。这个秩函数的表现有点像一个理想化的高度：它依赖于以 x 为根的子树的大小，而不是子树的高度，但如果子树完全平衡，则它指示子树的高度是多少。

如果 x 是一个树叶，则 $|T_i(x)|=1$，所以 $r_i(x)=0$。靠近树的边缘的结点有较小的秩，根具有树中最大的秩。

算法在子树中执行插入或删除必须的工作量明显与子树的高度有关，因此，我们希望它与子树根的秩有关。由于伸展中的工作量随着树的高度增加，因此我们打算以这样的方式定义存款余额：较大和较高的树将具有较大的存款余额，较短或较小的树具有较小的余额。我们将使用秩函数去实现它，事实上，我们将在树的所有顶点之间分配存款余额，总是要求下列条件成立：

* * * 银行存款不变量

银行存款不变量

对 T 的每一个结点 x 和伸展后的每一步 i，
结点 x 有与它的秩 $r_i(x)$ 相等的银行存款。

* * * 总共的存款余额

树的总共的存款余额简单地定义为树中所有结点各自银行存款的和：

$$c_i = \sum_{x \in T_i} r_i(x)$$

如果树是空的或仅包含一个结点，则它的存款余额是 0。随着树的增长，它的存款余额增加，并且这个余额反映了建立这棵树所需要的工作，树中银行存款的投资以两种方式完成：

- 投资操作中所做的实际工作。我们已决定将目标结点在伸展过程中提升的每一层作为一个单元来计算，因此除了 zig 或 zag 步骤计作一个单元外，每个伸展步骤计作两个单元。
- 既然树的形状在伸展期间改变，为了在所有时间内维护银行存款不变量，我们必须增加或者删除在树中投资的银行存款。（就如我们在最近一节中讨论的，这实质上是一个使不同步骤的代价平坦的记账工具。）

＊＊＊平摊复杂性

定义第 i 步的平摊复杂性 a_i 的等式概括了这个投资：$a_i = t_i + c_i - c_{i-1}$，其中 t_i 是实际工作，$c_i - c_{i-1}$ 给出了存款余额的变化。

＊＊＊主要目标

我们现在的主要目标使用给定的定义和伸展方式去确定 a_i 的边界，它反过来又将使我们发现整个伸展过程的代价，这个代价平摊在一系列的检索和插入上。

首先，我们需要一个预备的数学观察资料。

引理 10.6　如果 α，β 和 γ 是正实数，且 $\alpha+\beta \leqslant \gamma$，则 $\lg\alpha + \lg\beta \leqslant 2\lg\gamma - 2$。

证明　因为任何实数的平方是非负数，因此我们有 $(\sqrt{\alpha} - \sqrt{\beta})^2 \geqslant 0$，它经扩展和简化后得到

$$\sqrt{\alpha\beta} \leqslant \frac{\alpha+\beta}{2}。$$

＊＊＊算术－几何平均不等式

（这个不等式称为算术－几何平均不等式。）因为 $\alpha+\beta \leqslant \gamma$，我们得到 $\sqrt{\alpha\beta} \leqslant \gamma/2$。在两边平方并取（底为 2 的）对数即得到引理中的结果。（**证明结束**）

接下来我们分别分析各种不同的伸展步骤。

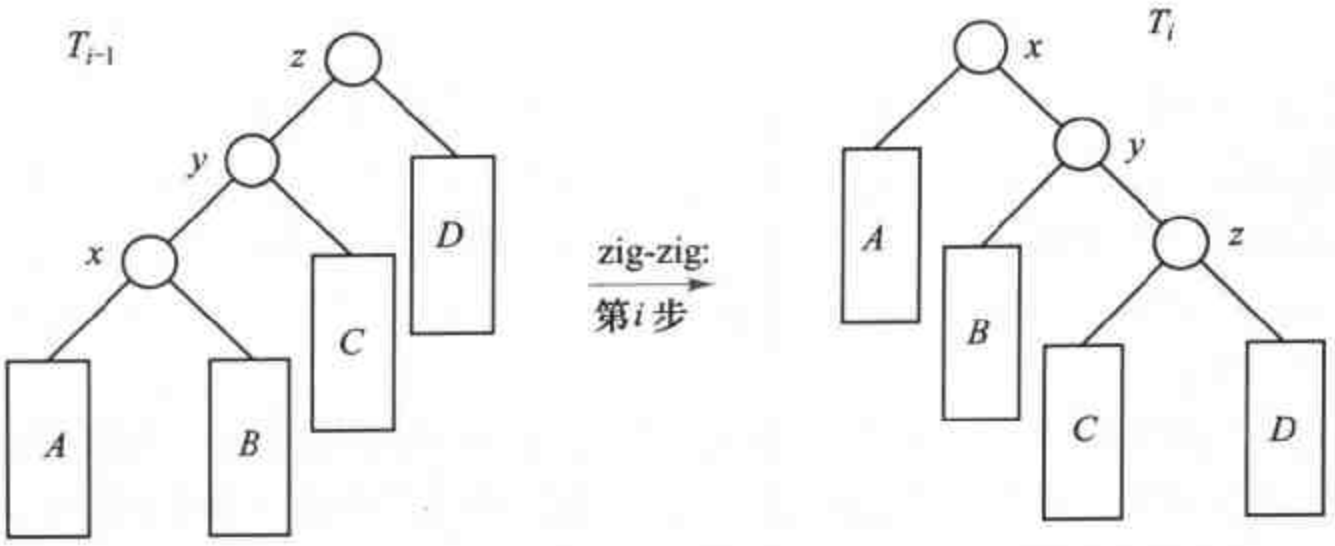
引理 10.7　如果第 i 步伸展是在结点 x 上的 zig-zig 或 zag-zag，则它的平摊复杂性 a_i 满足 $a_i < 3(r_i(x) - r_{i-1}(x))$。

证明　这种情况如下图（见下页）：

zig-zig 或 zag-zag 步骤的实际复杂性 t_i 是 2 个单元，而且仅仅以 x，y 和 z 为根的子树，它的大小会在这一步改变，因此，除了下列等式中表示的那些项外，总和中定义 c_i 的所有项抵消了 c_{i-1} 的那些项：

$$\begin{aligned} a_i &= t_i + c_i - c_{i-1} \\ &= 2 + r_i(x) + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) - r_{i-1}(z) \\ &= 2 + r_i(y) + r_i(z) - r_{i-1}(x) - r_{i-1}(y) \end{aligned}$$

通过取式子 $|T_i(x)| = |T_{i-1}(z)|$ 的对数后得到最后一行，从这个式子可以观察到伸展步骤之前以 z 为根的子树与这个步骤后以 x 为根的子树具有同样的规模。



现在能应用引理 10.6 去消除这个等式中的 2(实际复杂性)。设 $\alpha=|T_{i-1}(x)|$,$\beta=|T_i(z)|$,$\gamma=|T_i(x)|$。从这个情况的图中,我们看到 $T_{i-1}(x)$包含成分 x,A 和 B;$T_i(z)$包含成分 z,C 和 D;$T_i(x)$包含所有这些成分(y 除外)。因此 $\alpha+\beta<\gamma$,所以引理 10.6 蕴含了 $r_{i-1}(x)+r_i(z)\leqslant 2\ r_i(x)-2$,或 $2\ r_i(x)-r_{i-1}(x)-r_i(z)-2\geqslant 0$。将这个非负量加到上一个 a_i 等式的右边,我们得到 $a_i\leqslant 2\ r_i(x)-2\ r_{i-1}(x)+r_i(y)-r_{i-1}(y)$。

在步骤 i 前,y 是 x 的双亲,因此$|T_{i-1}(y)|>|T_{i-1}(x)|$。在步骤 i 后,x 是 y 的双亲,因此$|T_i(x)|>|T_i(y)|$。取对数时,我们有 $r_{i-1}(y)>r_{i-1}(x)$和 $r_i(x)>r_i(y)$。因此我们最后得到 $a_i<3\ r_i(x)-3\ r_{i-1}(x)$,这是引理 10.7 中我们希望证明的断言。(**证明结束**)

引理 10.8 如果第 i 个伸展步骤是在结点 x 的 zig-zag 或 zag-zig,则它的平摊复杂性 a_i 满足 $a_i<2(\ r_i(x)-r_{i-1}(x))$。

引理 10.9 如果第 i 个伸展步骤是在结点 x 的 zig 或 zag,则它的平摊复杂性 a_i 满足 $a_i<1+(\ r_i(x)-r_{i-1}(x))$。

引理 10.8 的证明类似于引理 10.7 的证明(即使结果更强),引理 10.9 的证明是直接的,这两个引理的证明都留作练习。

最后,我们需要找到检索或插入的总共的平摊代价。为此,我们必须将检索或插入期间所做的所有伸展步骤的代价相加,如果有 m 个这样的步骤,则至多(最后)一个步骤可以是 zig 或 zag,我们对它应用引理 10.9,而其他的都满足引理 10.7 和 10.8 中的范围,引理 10.7 中的系数 3 提供了更弱的范围。因此我们得到总的平摊代价是:

$$\begin{aligned}\sum_{i=1}^{m} a_i &= \sum_{i=1}^{m-1} a_i + a_m \\ &\leqslant \sum_{i=1}^{m} (3r_i(x)-3r_{i-1}(x))+(1+3r_m(x)-3r_{m-1}(x)) \\ &= 1+3r_m(x)-3r_0(x) \\ &\leqslant 1+3r_m(x) \\ &= 1+3\lg n\end{aligned}$$

* * *套叠和

在这个推导里,使用了套叠和的事实,从而只保留第一个秩 $r_0(x)$和最后一个秩 $r_m(x)$。

由于 $r_0(x) \geqslant 0$，它的省略只增加了右边，而且因为在伸展过程结束时 x 是树的根，因此我们有 $r_m(x)=\lg n$，这里 n 是树中的结点数。

以此，我们即完成了这一节主要结果的证明：

定理 10.10　在一棵具有 n 个结点的二叉查找树中，带伸展的插入或检索的平摊代价不会超过 $1+3\lg n$ 次目标结点在树中向上的移动。

最后，我们能将此平摊代价与较长系列的伸展插入或检索中的每一个实际代价联系起来。为此，我们应用引理 10.5，注意到现在是对一系列检索或插入求和，而不是在单独的检索或插入步骤上。我们看到一系列 m 个伸展访问的总共的实际代价与总共的平摊代价的不同之处仅仅是相差 c_0-c_m，其中 c_0 和 c_m 分别是初始和终了树的存款余额。如果树决不会超过 n 个结点，则任何单独结点的银行存款总是 0 和 $\lg n$ 间的某个值。为此树的初始和终了平摊代价都在 0 和 $\lg n$ 之间，因此我们至多需要在定理 10.10 中的代价上再增加 $n\lg n$，得到：

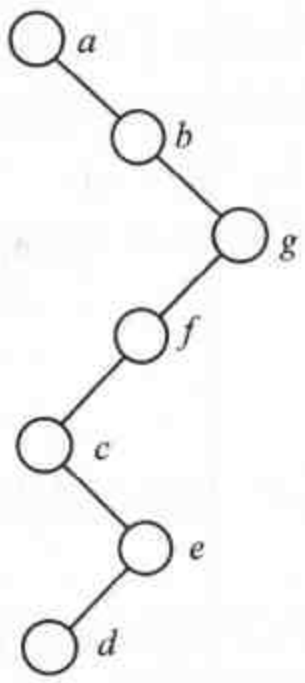
推论 10.11　在一棵结点数不会超过 n 的二叉查找树中，一系列 m 个带伸展的插入或检索的总复杂性不超过 $m(1+3\lg n)+n\lg n$ 次目标结点在树中向上的移动。

在这个结果中，除了 zig 或 zag 计作一次移动外，每个伸展步骤都计作两次上移。

假设在任何时候，伸展树都有可能退化到高度不平衡的形状，在伸展树中，较长系列的插入和检索保证仅花费 $O(\log n)$ 时间的事实是很引人注目的。

练习 10.5

E1. 考虑下面的二叉查找树：



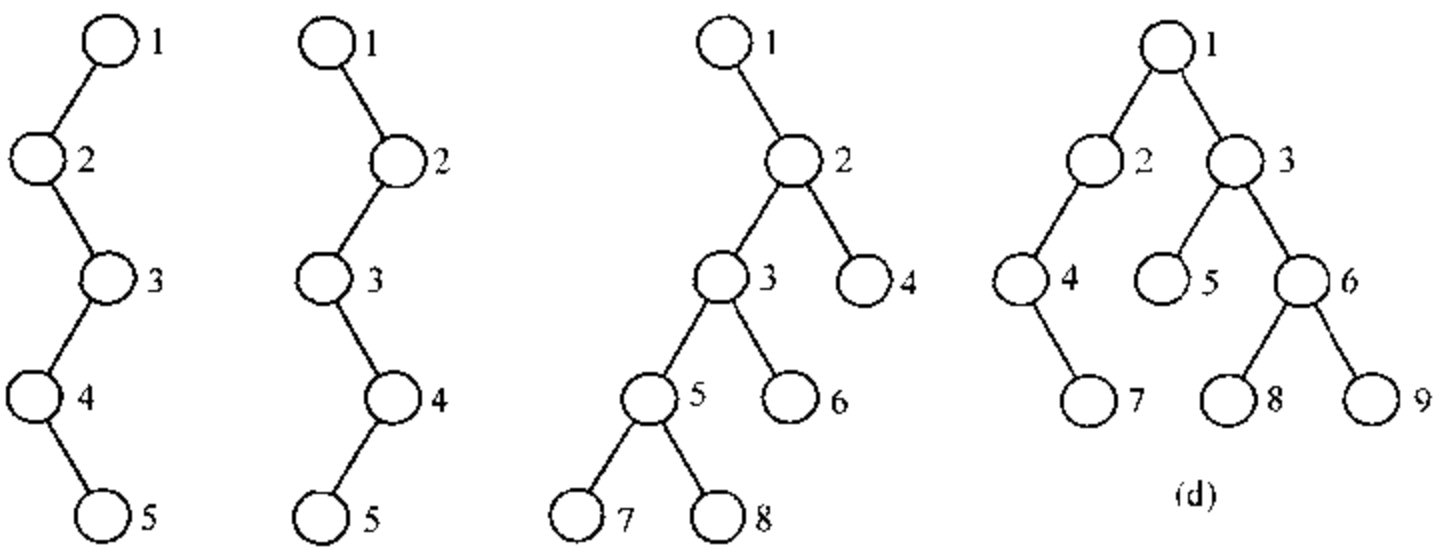
依次在下列每个键上伸展这棵树：

$$d \quad b \quad g \quad f \quad a \quad d \quad b \quad d$$

每个部分都建立在以前的基础上，也就是说，将每个解答得到的最终树作为下一部分的起始树。（检查：在最后一部分之后，与前一部分一样，树应当完全平衡。）

E2. 二叉树中结点的深度是从根到这个顶点的分支数。（这样，根的深度为 0，它的孩子深度为 1，以此类推。）前序遍历期间树的存款余额定义为被访问顶点的深度；访问一个顶点

的(实际)代价定义为从前一受访顶点开始(或向上或向下)遍历的分支数。对下列每一棵二叉树,建立一个表格去说明被访问的顶点、实际的代价、存款余额和前序遍历的平摊代价。



E3. 如下定义任意二叉树的结点的秩函数 $r(x)$:

如果 x 是根,则 $r(x)=0$。

如果 x 是结点 y 的左孩子,则 $r(x)=r(y)-1$。

如果 x 是结点 y 的右孩子,则 $r(x)=r(y)+1$。

遍历期间树的存款余额定义为被访问结点的秩;访问一个顶点的(实际)代价定义为从前一访问结点出发(或向上或向下)所遍历的分支数。对练习 E2 中所示的每一棵二叉树,建立一个表格以说明访问的顶点、实际的代价、存款余额和中序遍历的平摊代价。

E4. 与 E3 类似,为二叉树设计一个秩函数,在与练习 E3 同样的条件下,采取类似步骤完成后序遍历的平摊代价。为练习 E2 中所示的每一棵二叉树建立一个表格,说明访问的顶点、实际的代价、存款余额和后序遍历的平摊代价,以此证明你的秩函数,

E5. 将课文中递增 4 位数字的二进制整数给出的平摊分析推广到 n 位数字的二进制整数。

E6. 证明引理 10.8,方法类似于引理 10.7 的证明。

E7. 证明引理 10.9,这个证明不需要引理 10.6 或任何复杂的计算。

程序设计项目 10.5

P1. 将伸展树类替换到 10.2 节项目 P2 中用于二叉查找树的菜单驱动演示程序中,因而得到伸展树的演示程序。

P2. 用伸展检索和插入替换 10.2 节项目 P5 的信息检索项目,对各种组合的输入文本文件,比较伸展树的性能和普通二叉查找树的性能。

启示和易犯的错误

1. 将二叉查找树看作是有序表的一种选择(实际上是作为实现抽象数据类型 list 的一种方式)。二叉查找树以每个结点中的额外指针成员为代价,在保持链表插入、删除和重排的

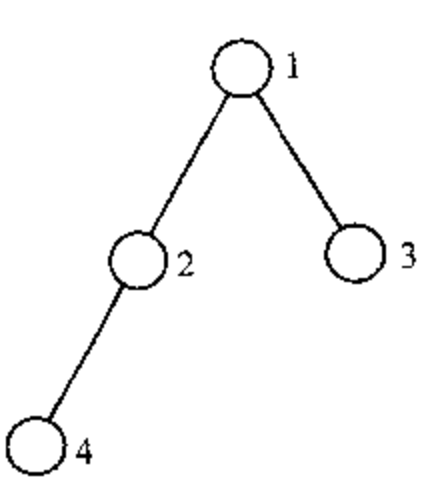
灵活性的同时，允许到树中所有结点的随机访问(($O(\log n)$次键的比较)。

2. 将二叉查找树看作是表格的一种选择(实际上是作为实现抽象数据类型 table 的一种方式)。二叉查找树以 $O(\log n)$而不是 $O(1)$的访问时间为代价，在保持表格提供的随机访问的优点的同时，允许以键指定的次序遍历数据结构。
3. 在选择数据结构时，总是仔细考虑所需要的操作。当随机访问、以预定的次序遍历、插入和删除的灵活性等要求都需要满足时，二叉树尤其合适。
4. 当选择数据结构和算法时，警惕高度不平衡的二叉查找树的可能性。如果进来的数据可能是以随机次序，则普通的二叉树证明完全足够用。如果数据可能以有序或接近有序的次序进来，则算法应采用适宜的操作。如果只存在略微不平衡的可能性，则可以忽略。而如果在一个大型项目中存在更大可能的严重不平衡，则软件中要在合适的位置检查树的平衡性，必要时要重建树的平衡性。对于所有时间内都必须保持对数访问时间的应用，AVL 树以较小的计算机时间和空间代价提供了接近完美的平衡，但它却要求付出可观的程序设计代价。如果树必须动态适应数据频繁的改变，则伸展树可能是最好的选择。
5. 二叉树采用递归定义，因此操作二叉树的算法一般最好采用递归编写。在用二叉树进行程序设计时，要意识到问题通常与递归算法有关。确信算法在任何条件下都会终止，而且算法正确地处理了空树的简单情况。
6. 虽然二叉树通常以链式结构实现，但也要意识到仍存在其他实现的可能性。在用链式二叉树进行程序设计时，谨记用链表进行程序设计时易犯的错误。

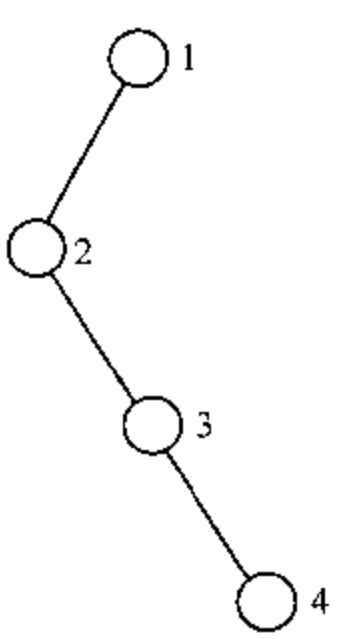
复习题

10.1

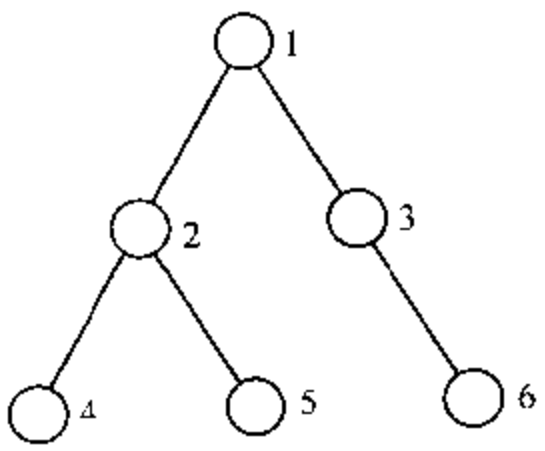
1. 定义术语二叉树。
2. 二叉树和顶点至多有两个分支的普通树之间的区别是什么?
3. 给出以(1)前序；(2)中序；(3)后序遍历下列每棵二叉树时的顶点访问次序。



(a)



(b)



(c)

4. 对下列的每个表达式，画出表达式树，说明以(1)前序，(2)中序和(3)后序遍历树的结果。

(1)$a-b$

(2)$n/m!$

(3)$\log m!$

(4)$(\log x)+(\log y)$

(5)$x\times y\leqslant x+y$

(6)$(a>b)\ \|\ (b>=a)$

10.2

5. 定义术语二叉查找树。
6. 如果 n 个结点的二叉查找树已很好地平衡,则找到一个目标所需要的近似键比较次数是多少?如果这棵树退化成一条链,则次数是多少?
7. 用 20 个或更少的词,解释树排序如何工作。
8. 树排序和快速排序之间的关系是什么?
9. 什么原因导致从查找树中删除比插入到查找树中更加困难?

10.3

10. 在 10.3 节中设计的建立二叉查找树的算法何时有用?对输入中的每一项,简单地用这个函数将一项插入到查找树中为何更可取?
11. 平均来说,查找一棵随机二叉查找树比查找完全平衡二叉查找树慢多少?

10.4

12. AVL 树的目的是什么?
13. 在所有二叉查找树中定义 AVL 树的条件是什么?
14. 假设 A 是一个基类,B 是派生类,并且定义:A * pA;B * pB;则 pA 能引用 B 类的对象吗?pB 能引用 A 类的对象吗?
15. 解释类的虚方法与其他的类方法有何不同?
16. 画图解释当插入到 ALV 树中导致结点失去平衡时,如何恢复平衡。
17. AVL 树的最坏性能与随机二叉查找树的最坏情况性能相比如何?与它的平均情况性能相比如何?AVL 树的平均情况性能与随机二叉查找树的平均情况性能相比如何?

10.5

18. 用 20 个或更少的词,描述伸展的作用?
19. 伸展的目的是什么?
20. 什么是平摊算法分析?
21. 什么是存款余额函数?如何使用它?
22. 在大-O 符号中,伸展平摊在一系列检索和插入上的代价是什么?为何结果令人吃惊?

进阶参考书目

关于二叉树最全面的信息来源之一是 Knuth 的系列书,二叉树的特征、其他类的树、遍历、路径长度和历史信息,在第 1 卷的第 305～405 页。第 3 卷的第 422～480 页讨论了二叉查找树、AVL 树和相关的主题。定理 10.2 的证明来自第 3 卷 427 页,

关于 AVL 树性能的数学分析出现于:E. M. Reingold, J. Nievergelt, and N. Deo,

Combinatorial Algorithms: Theory and Practice, Prentice Hall, Englewood Cliffs, N. J. 1997.

下面的书提供了关于各种数据结构的许多有趣的实验研究和其他分析，包括二叉查找树：Robert Sedgewick and Philippe Flajolet, *An Introduction to the Analysis of Algorithm*, Addison-Wesley, Reading, Mass., 1996.

关于AVL树的原始参考书目是：G. M. Adel′Son-Vel′Skii and E. M. Landis, *Dokl. Akad. Nauk SSSR* 146 (1962), 263-266; 英文译本: *Soviet Math.* (*Dokl.*) 3 (1962), 1259-1263.

下面这本书中讨论了构造平衡二叉树的几个算法：Hsi Chang and S. S. Iyengar, "Efficient algorithm to globally balance a binary search tree", *Communications of the ACM* 27 (1984), 695-702.

伸展树和平摊算法分析的概念，以及我们提供的算法的出处属于：D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees", *Journal of the ACM* 32 (1985), 652-686.

介绍与本章有关的更高级主题的较好来源是：Harry R. Lewis and Larry Denenberg, *Data Structure & Their Algorithms*, Harper-Collins, New York, 1991, 509 pages. 和 Derick Wood, *Data Structures, Algorithms, and Performance*, Addison-Wesley, Reading, Mass., 1993, 594 pages.

调整二叉树查找形状的另一种有趣的方法称为带权路径树，它基于结点被访问的频度，此方法出现在下面这篇文章中，易于阅读，并且还有对相关结果的概论：G. Argo, "Weighting without waiting: the weighted path length tree", *Computer Journal* 34 (1991), 444-449.

第11章　多　路　树

本章继续学习树这种数据结构，重点放在每个结点可能有多于两个分支的树上。我们首先建立与二叉树的连接，接下来研究一类称为trie的树，它具有表查找的某些特征。然后我们研究B-树，它对于外部信息检索问题非常有价值。上面这几节是互相独立的。最后，我们应用B-树的思想得到另一类二叉查找树，称为红-黑树。

11.1　果园、树和二叉树

就如我们所看到的，二叉树是一种功能强大、形态优雅的数据结构。即便如此，对每个结点不超过两个孩子的限制还是显得太严格，可能有许多以树作为数据结构的应用，其结点的孩子数是任意的。本节说明一个合意且有用的惊奇之处：二叉树提供了一种便利的方法来表达初看起来非常广泛的一类树。

11.1.1　物种分类

＊＊＊数学定义

我们在刚学过的应用中已见到了几种树，因此在进一步研究之前，我们首先解决它的定义。在数学中，树这个术语有着相当广泛的含义：它是点（称为顶点）的任意集合和不同顶点对（称为边或分支）的任意集合，使得：(1)从任一顶点到任意其他顶点有一个由边组成的序列（路径），并且(2)没有回路，即没有从一个顶点开始并返回到同一顶点的路径。

＊＊＊自由树

＊＊＊有根树

在计算机应用中，我们通常不需要研究这样太一般的树，而如果需要时，为强调起见，我们称它们为自由树（free tree）。我们几乎总是在树中挑出一个特殊的顶点作为树根，从而对树进行一定的约束，为强调起见，我们称这样的树为有根树（rooted tree）。

＊＊＊有序树

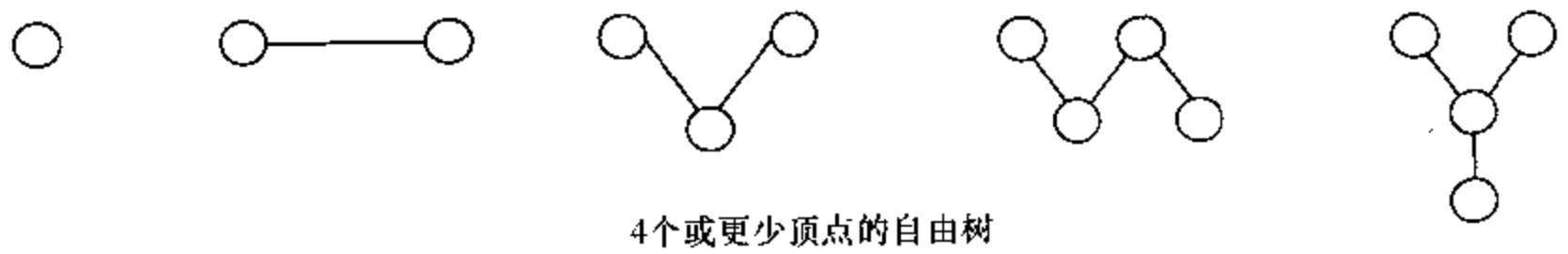
我们能以通常的方法画出有根树：提起它的根并晃动它，这样使得所有的分支和顶点都向下悬挂，树叶在底部。即便如此，有根树并不具有我们通常使用的所有结构。在有根树中，我们无法区分左右，或者当顶点有几个孩子时，我们无法区分谁是第一个，谁是第二个，等等。如果没有其他原因，指令的顺序执行（不是指顺序的存储组织）的约束通常在每个顶点的孩子上强加了一个次序。因此我们将有序树（ordered tree）定义为每个顶点的孩子分配了一个次序的有根树。

注意：没有一个顶点有多于两个孩子的有序树与二叉树仍不属于同一类。如果二叉树中的一个顶点仅有一个孩子，则这个孩子可以在该顶点左边也可以在其右边，而结果得到的两棵二叉树是不同的，但这二者都与有序树相同。

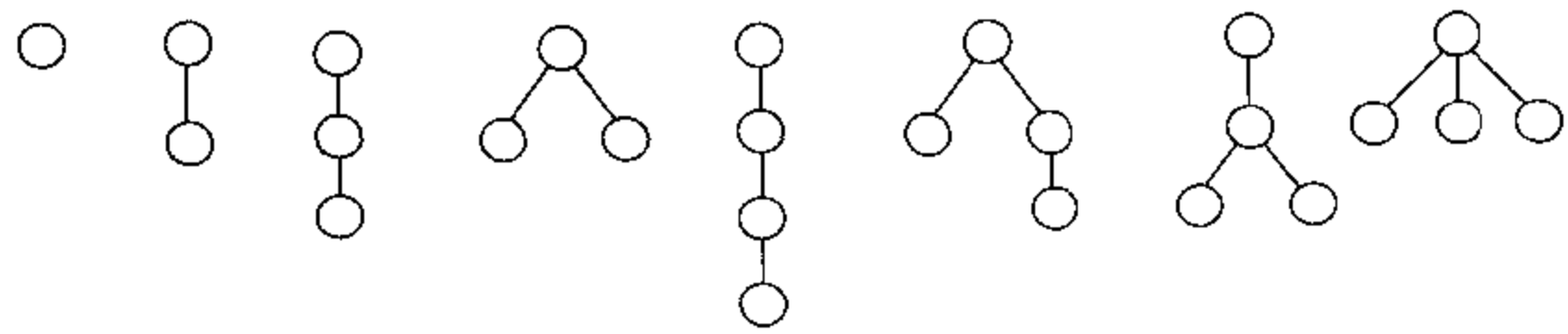
＊ ＊ ＊2－树

作为最后与定义有关的备注，我们指出：曾作为算法分析的一部分学习的 2－树是有根树(但不必是有序树)，它的特征是每个顶点有 0 或 2 个孩子。因此 2－树与我们已介绍的任何其他类的树都不相符。

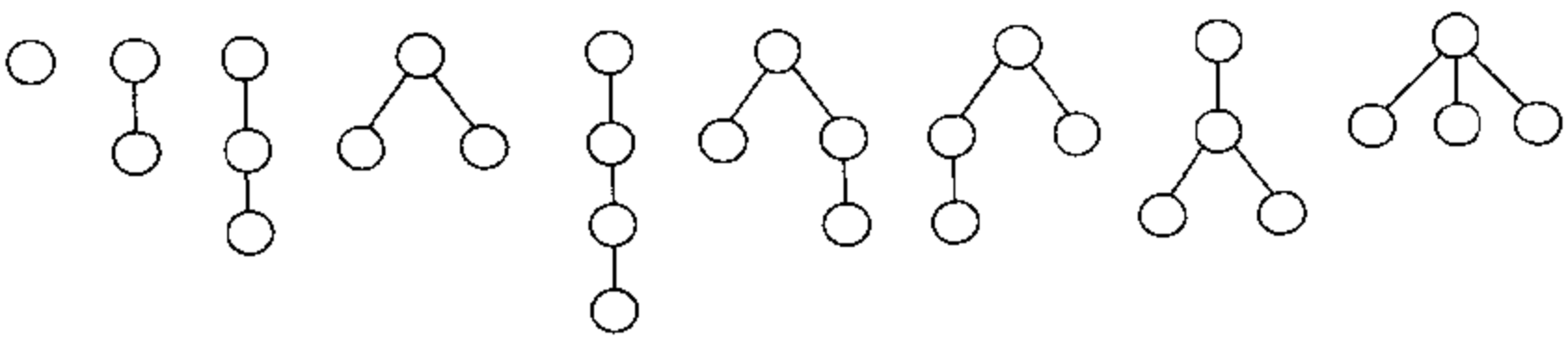
图 11.1 显示了具有较少结点的各种树的情况。注意第一棵树之后的每种树都能通过前面种类的树得到，而且因新准则造成的不同而能够加以区分。将具有 4 个顶点的 5 棵有序树与 10.1 节练习 E1 中构造的具有 4 个顶点的 14 棵二叉树进行比较，你将再次发现，通过区分左分支和右分支，可以从合适的有序树得到二叉树。



4个或更少顶点的自由树
(与顶点的排列无关)



4个或更少顶点的有根树
(根在树的顶部)



4个或更少顶点的有序树

图 11.1　各种各样的树

11.1.2　有序树

1. 计算机实现

＊ ＊ ＊ 多链

如果我们希望将有序树作为数据结构，在计算机内存中实现它的明显方法是扩展实现二叉树的标准方法，为每个结点保留与可能有的子树同样多的链，而不仅是二叉树所需要的两个链。因此在一棵某些结点有 10 棵子树的树中，我们将为每个结点保留 10 个链成员。

但这会导致大量的链成员为空。实际上，我们能简单精确地确定有多少链。如果一棵树有 n 个结点，且每个结点有 k 个链成员，则总共有 $n*k$ 个链。除了根以外，只有一条链指向 $n-1$ 个结点中的每一个结点，因此空链的比例必然为：

$$\frac{(n\times k)-(n-1)}{n\times k}>1-\frac{1}{k}$$

* * * 浪费的空间

因此如果一个顶点可能有 10 棵子树，则超过 90%的链将为空。显然，这种表示有序树的方法非常浪费空间。其原因是，对每个结点，我们都保持它对所有孩子链一个顺序表，而此顺序表中留有大量未用的空间。我们现在研究一种方法，采用链表来代替这些顺序表，从而与二叉树形成优美的衔接。

2. 链式实现

* * * first_child 链

* * * next_sibling 链

为了将每个结点的孩子保存在一个链表中，我们需要两种链。第一种链位于一个孩子家庭的头部，它是从双亲结点到其最左孩子的一个链，我们可以称它为 first_child。第二种链除根以外的每个结点都会出现于这些表中的某一个内，因此需要一个链指向表中的下一个结点，也就是指向双亲的下一个孩子，我们可以称这第二种链为 next_sibling。图 11.2 说明了这种实现方法。

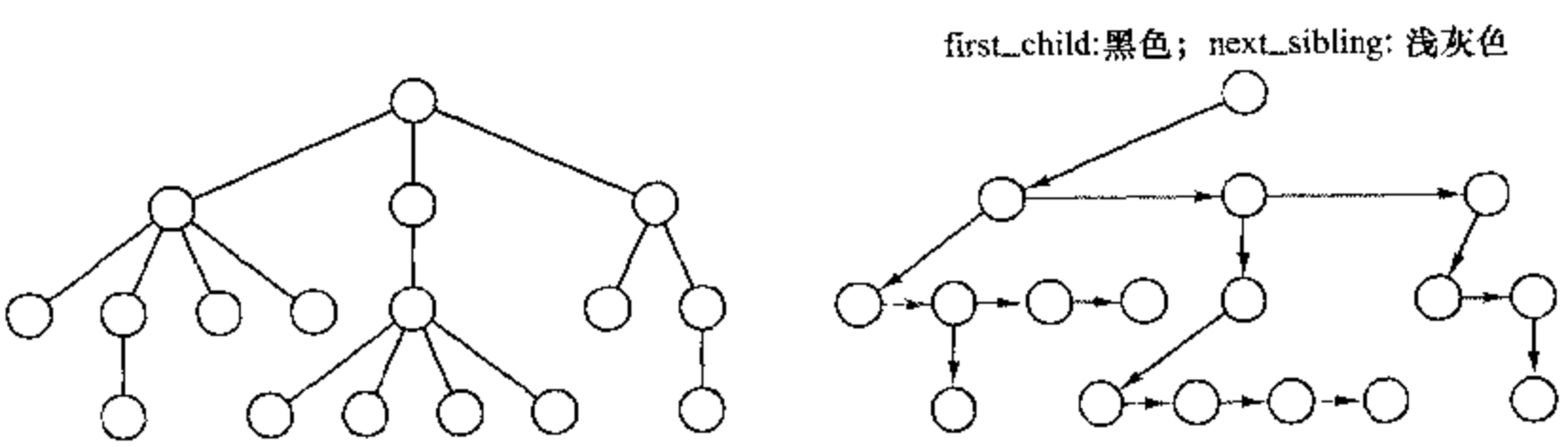


图 11.2 有序树的链式实现

3. 自然对应

对有序树的每个结点，我们已定义了两个链 first_child 和 next_sibling(否则如果没有定义的话则为空)。通过使用这两个链，有序树现在具有二叉树的结构，即有序树的链式实现是链式二叉树。如果我们愿意，甚至可以形成二叉树的一幅更美好的图画：采用有序树的链式表示，并将它按顺时针方向旋转一些角度，使得向下的(first_child)链指向左边，而水平的(next_sibling)链指向右下方。因此，对图 11.2 中的树，我们得到图 11.3 中的二叉树。

4. 反转对应

假设将前面过程的步骤反转，由二叉树开始，试着恢复一棵有序树。首先必须观察：并非每棵二叉树都可以由有根树通过前述过程得到；因为根的 next_sibling 链总是为空，相应

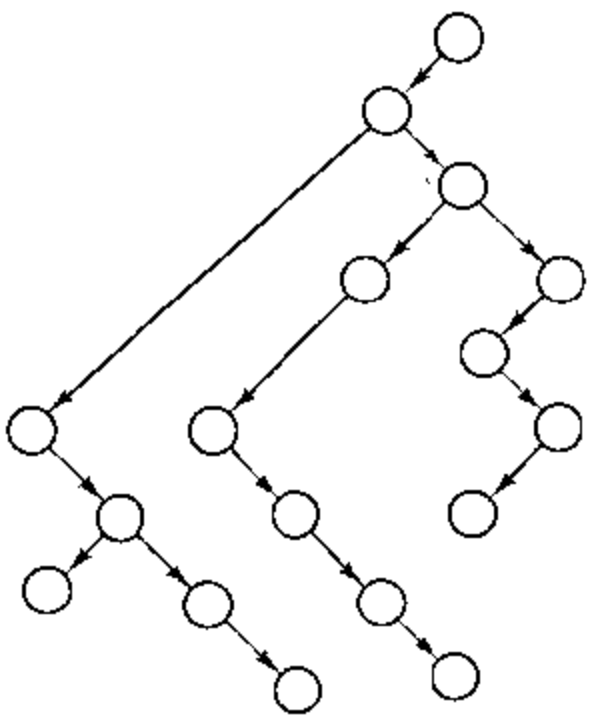


图 11.3　链式实现旋转后的形式

二叉树的根将总是有一棵空的右子树。为了更仔细地研究反转对应，我们必须考虑另一类数据结构。

11.1.3　森林和果园

迄今对二叉树的工作，我们都得益于递归，且对其他种类的树，我们将继续使用递归。运用递归意味着将问题简化得更小。因此如果采用有根树或有序树且剥落根，我们将看到所发生的事情。如果左边是(若不空)有根树的集合或有序树的有序集合，则分别会发生什么?

* * * 森林、果园

对任意集合的树，可以使用的标准术语是森林，但当我们使用这个术语时，通常会假设树是有根的。有时候我们也使用有序森林这个术语描述有序树的有序集合，但对它我们将采用具有同样描述能力(更有色彩)的术语果园。

注意我们不仅可以通过分别删除有根树或有序树的根得到森林或果园，而且能够通过从森林或果园开始，在顶部附加新结点，从新顶点(将作为根)增加分支到森林或果园中的所有树，从而构造有序树或有根树。图 11.4 说明了这些操作。

* * * 递归定义

我们将使用这个过程给出有序树和果园的新的递归定义，它也形成了与二叉树衔接的形式证明。首先，考虑如何开始。回忆一下，二叉树可能为空，也就是，它将没有顶点。森林或果园也有可能为空，即它们不包含树。然而，有根树或有序树不可能为空，因为它至少保证包含根。如果我们希望开始构建树和森林，就能注意到仅有一个顶点的树可以通过将一个新根附加到一个空的森林中而得到。一旦有了这棵树，就能形成森林，此森林由具有单个顶点的树组成，树的数目如我们给定的，随后我们能附加一个新根去建立所有高度为 1 的有根树。以这种方式，依照下列互相递归的定义，我们能继续依次构造出所有的有根树。

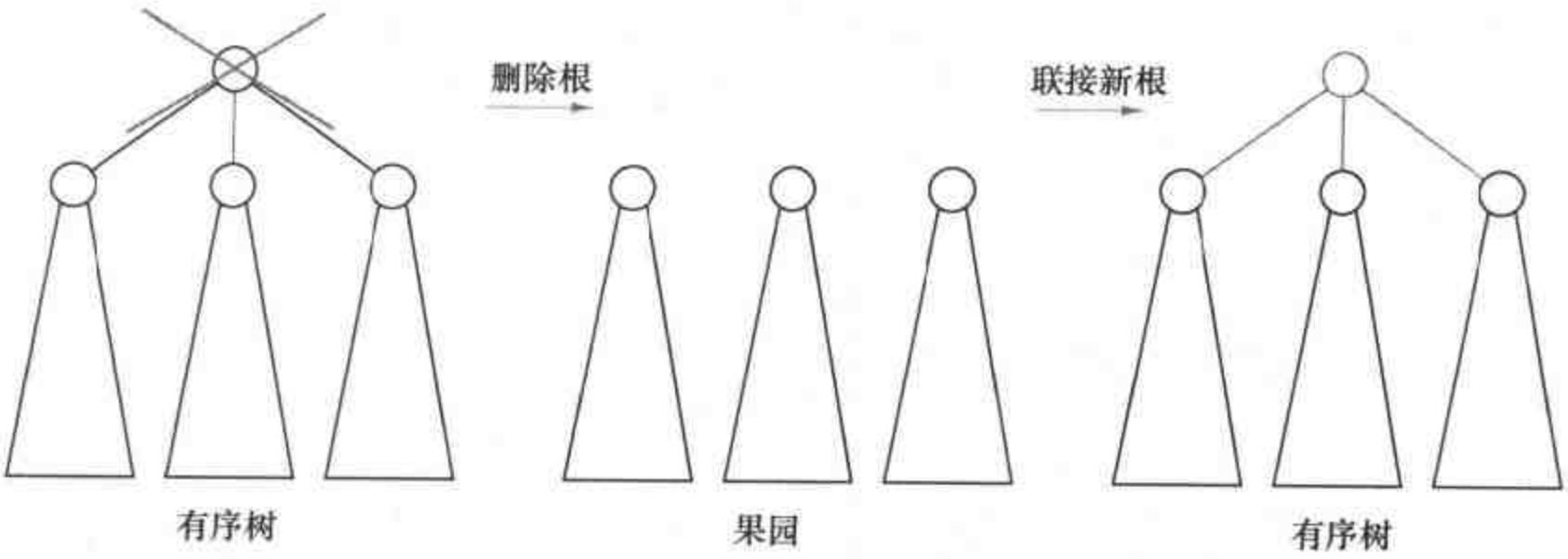


图 11.4 删除和联接根

定义 有根树由单个的称为树根的顶点 v 和森林 F 组成，森林的树称为根的子树。
森林 F 是(可能为空的)有根树的集合。

有序树和果园的构造与上类似。

定义 有序树 T 由单个的称为树根的顶点 v 和果园 O 组成，果园的树称为根 v 的子树。
我们能用下面的有序对表示有序树：$T=\{v,O\}$。
果园 O 或者是空集 Φ，或者由称为果园的第一棵树的有序树 T 加上另外的果园 O' (它包含了果园中剩余的树)组成。我们能用下面的有序对表示果园：$O=(T,O')$。

注意树的次序是如何蕴含在果园的定义中的。一个非空的果园包含第一棵树和来自另外果园的剩余树，这个果园又有第一棵树，而这第一棵树是最初果园的第二棵树。继续检查剩余的果园会产生第三棵树，以此类推，直到剩余果园为空，如图 11.5。

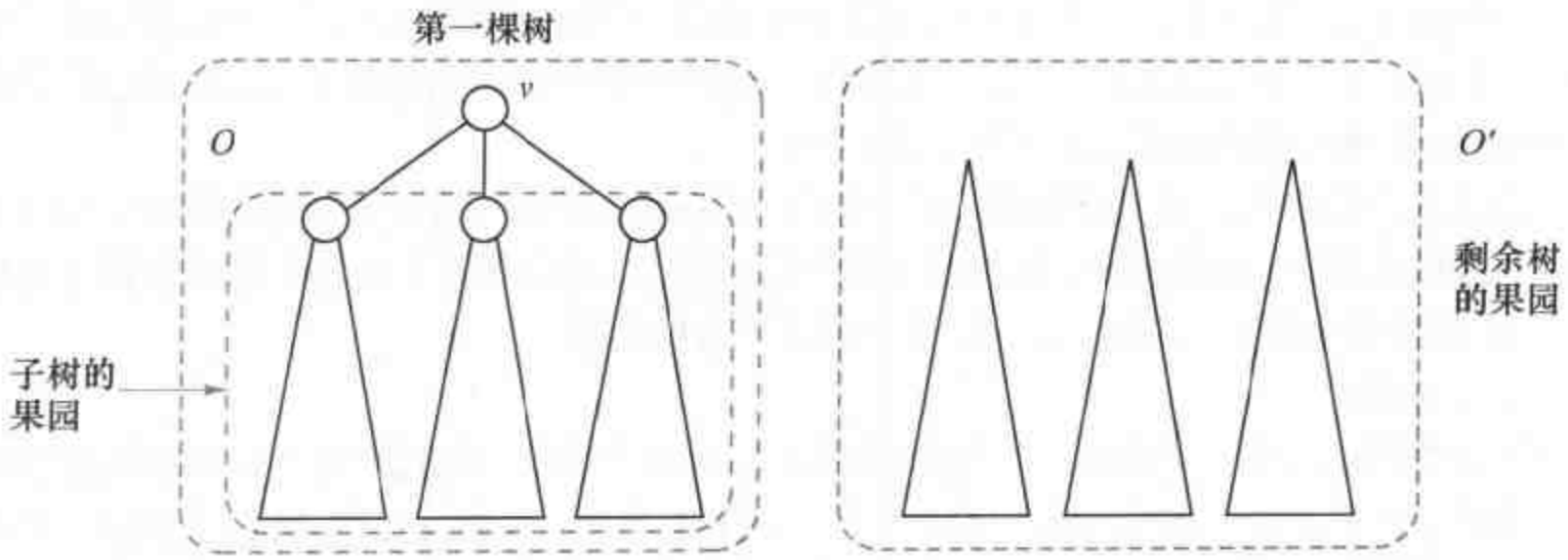


图 11.5 有序树和果园的递归构造

11.1.4 正式对应

我们现在能得到这一节的主要结果。

定理 11.1　设 S 是顶点的任一有限集。从顶点集为 S 的果园集到顶点集为 S 的二叉树集之间存在一一对应 f。

证明　我们使用定义中介绍的符号证明此定理。首先，我们需要用于二叉树的类似符号：二叉树 B 或者是空集 Φ 或者由根顶点 v 和 v 的两棵子树 B_1 和 B_2 组成。因此我们可以用下面的有序三元组表示非空二叉树：$B=[v,B_1,B_2]$。

我们将对 S 中的顶点数进行数学归纳来证明此定理。要考虑的第一种情况是空果园 Φ，它将对应于空二叉树：$f(\Phi)=\Phi$。如果果园 O 不为空，则它可以用有序对表示：$O=(T,O_2)$，这里 T 是有序树，O_2 是另外的果园。有序树 T 表示成 v 和 O_1 的对：$T=\{v,O_1\}$，这里 v 是一个顶点，O_1 是另外的果园。我们用该表达式替换第一个表达式中的 T，得到：$O=(\{v,O_1\},O_2)$。由归纳假设，f 提供了顶点少于 S 的果园到二叉树的一一对应，且 O_1 和 O_2 比 O 更小，因此，二叉树 $f(O_1)$ 和 $f(O_2)$ 由归纳假设确定。我们将果园到二叉树的对应 f 定义为：$f(\{v,O_1\},O_2)=[v,f(O_1),f(O_2)]$。显然函数 f 现在是果园和具有同样顶点的二叉树之间的一一对应。对任何填充到左边符号 v,O_1,O_2 的方法，只存在一种方法将同样符号填充到右边，反之亦然。（**证明结束**）

11.1.5　旋转

我们也能使用这个对应的符号形式帮助形成从果园到二叉树的变换图。在二叉树 $[v,f(O_1),f(O_2)]$ 中，来自于 v 的左链是到二叉树 $f(O_1)$ 的根，这实际上是有序树 $\{v,O_1\}$ 中 v 的第一个孩子。来自于 v 的右链是到先前作为右侧的下一个有序树的根的顶点。也就是说，二叉树中的“左链”对应有序树中的“第一个孩子”，“右链”对应“下一个兄弟”。可以用几何术语把此变换归纳为下列规则：

1. 画出果园，使每个顶点的第一个孩子直接位于该顶点下方，而不是将孩子集中画在该顶点下。
2. 从每个顶点到它的第一个孩子，画一条垂直的链，而从每个顶点到它的下一个兄弟画一条水平的链。
3. 删除剩余的初始链。
4. 将图顺时针旋转 45°，使得垂直的链显示为左链，水平的链显示为右链。

图 11.6 说明了这个过程。

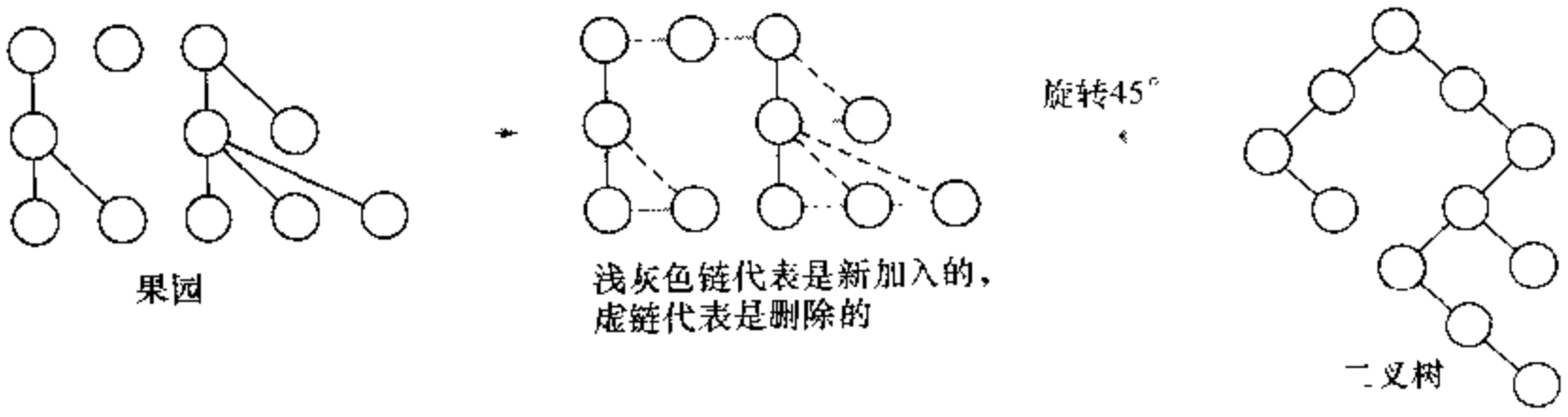


图 11.6　从果园到二叉树的转化

11.1.6 总结

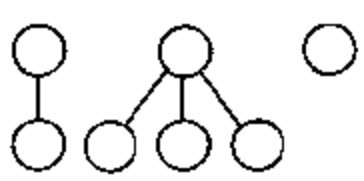
我们已了解描述果园和二叉树之间对应的3种方法。

- first_child 和 next_sibling 链；
- 图的旋转；
- 形式符号等价。

大多数人发现第二种方法，图的旋转，最易于记忆和绘画。第一种方法通过建立链给出对应关系，此方法通常为实际编写计算机程序所需要。最后，第三种形式对应的方法，对证明二叉树和果园的各种性质是最有用的。

练习 11.1

E1. 将下列每个果园转化成一棵二叉树。



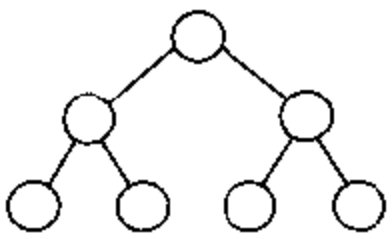
(a)



(b)



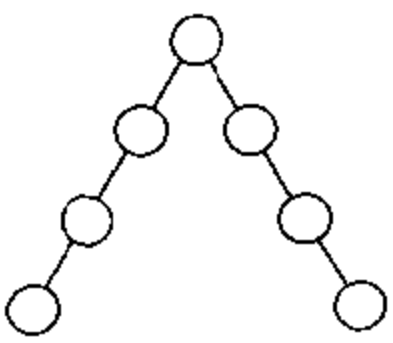
(c)



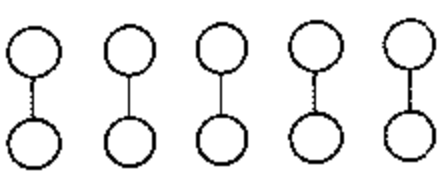
(d)



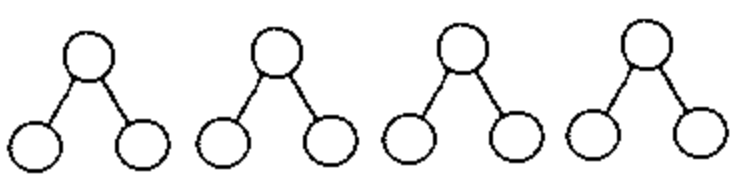
(e)



(f)

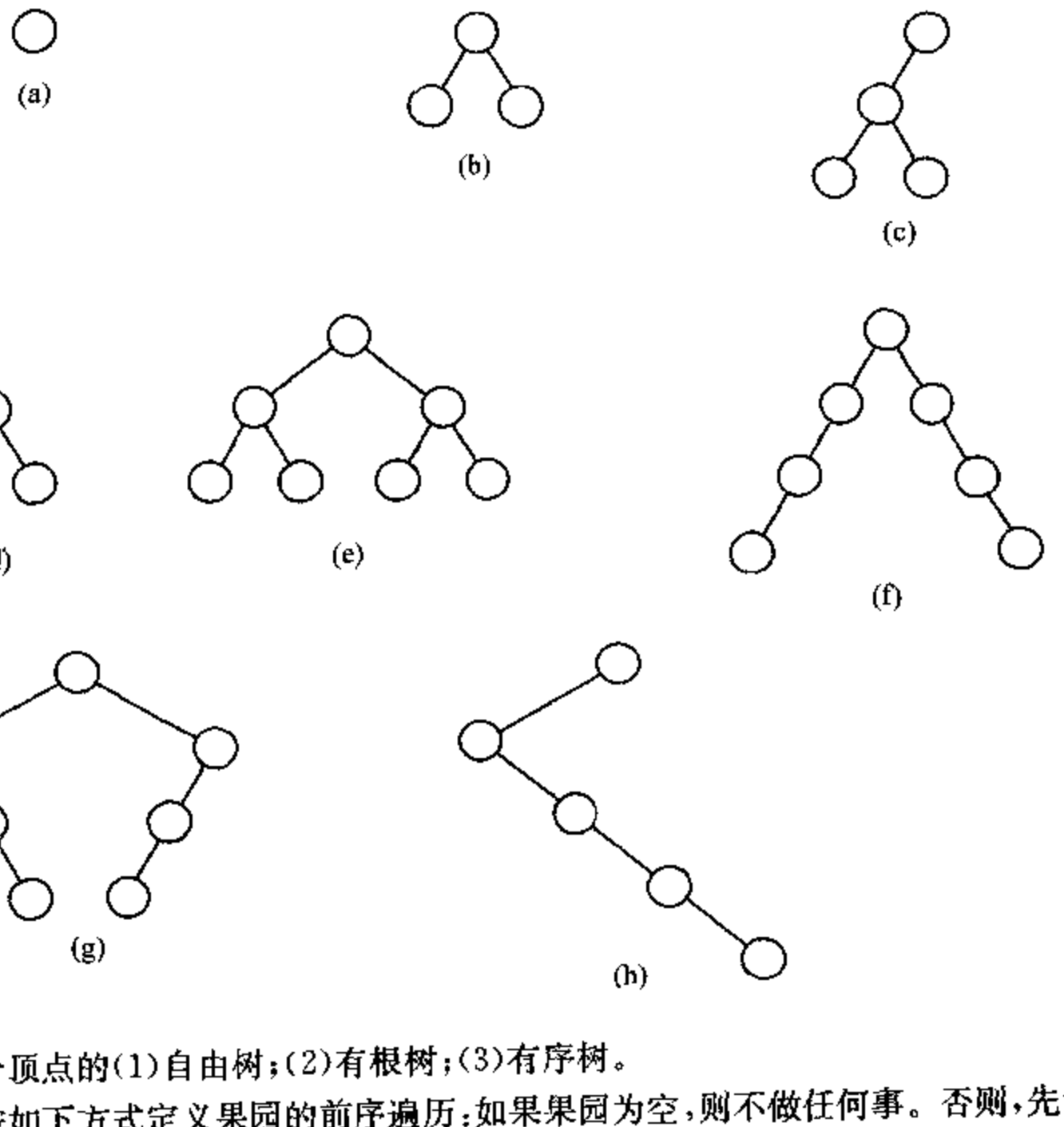


(g)



(h)

E2. 将下列每棵二叉树转化成一个果园。



E3. 画出 5 个顶点的(1)自由树;(2)有根树;(3)有序树。

E4. 我们能按如下方式定义果园的前序遍历:如果果园为空,则不做任何事。否则,先访问第一棵树的根,然后以前序遍历第一棵树的子树果园,然后以前序遍历剩余树的果园。证明果园的前序遍历和对应二叉树的前序遍历将以同样的次序访问顶点。

E5. 我们能按如下方式定义果园的中序遍历:如果果园为空,则不做任何事。否则,首先以中序遍历第一棵树的根的子树果园,然后遍历第一棵树的根,然后以中序遍历剩余子树的果园。证明果园的中序遍历和对应二叉树的中序遍历将以同样的次序访问顶点。

E6. 描述一种遍历果园的方法,它与对应二叉树的后序遍历以同样的次序访问顶点。证明你的遍历方法能以正确的次序访问顶点。

11.2　词典查找树:trie

＊＊＊多路分支

在前面章节中,已经多次将查找一个表和查找表格中的一个元素进行了对比。我们能将表格查找的思想运用到树中的信息检索,其方法是使用键或键的一部分作为多路分支。

我们能把键看作是一个字符序列(比如说,字母或数字),然后使用这些字符去确定每一步的多路分支,而不是利用完整键的比较。如果键是字母组成的名字,则我们根据名字的第一个字母安排 26 路分支,接下来根据第二个字母确定另一个分支,以此类推。这个多路分

支是词典中书边标目的思想。然而，书边标目通常仅用于根据给定的初始字母去查找词，然后再使用某些其他的方法继续。在计算机中，我们能通过多路分支推进两或三层，但树将变得很大，于是为了继续，我们将需要求助于其他的方法。

11.2.1 Trie

一种方法是从树中剪去所有不引至任何键的分支。比如，在英语中，没有词以字母'bb','bc','bf','bg'...开始，但有词以'ba','bd'或'be'开始。因此对不存在的词，能将所有的分支和结点从树中删除。结果得到的树称为 trie。(这个术语是从词 retrieval 中抽取字母而来，但通常发成词"try"的音。)

阶为 m 的 trie 能形式化地定义为或者是空树或者刚好由阶为 m 的 m 个 trie 的有序序列组成。

11.2.2 查找键

图 11.7 中显示的描述英语单词(如牛津英语词典中所列)的 trie 仅由字母 a,b 和 c 组

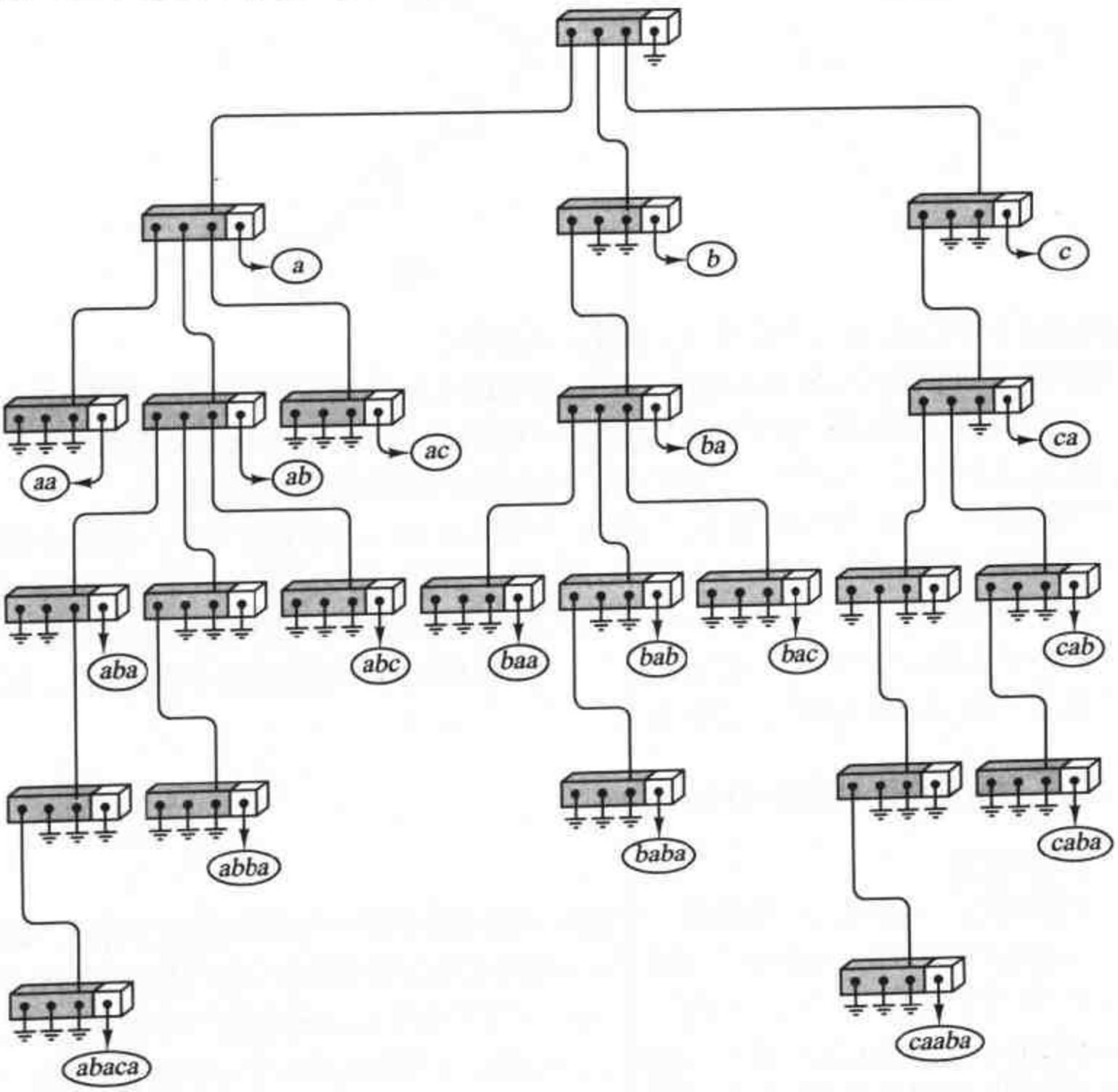


图 11.7 由 a,b,c 构成的词的 trie

成。连同到 trie 的下一层分支，每个结点包含一个指向关于键信息的记录指针，如果有的话，则当到达结点时就能发现它。对特定键的查找从根开始，键的第一个字母用于选择哪个分支的索引。空的分支意味着所查找的键不在树中。否则，我们使用键的第二个字母去确定下一层的分支，如此继续。当到达词的末尾时，信息指针将我们引至想要的信息。我们将使用 NULL 信息指针去表示字符串不是 trie 中的一个词。因此注意，词 a 是单词 aba 的前缀，aba 是词 abaca 的前缀。另一方面，字符串 abac 不是一个英语单词，因此它的结点有一个空的信息指针。

11.2.3 C++算法

我们将刚才描述的查找过程转变成查找以字符数组作为键的记录的方法。因此，我们将假设 Record 和 Key 类有 9.5 节描述的实现，在那儿对基数排序使用了类似的键：每个 Record 方法有一个 Key，它是字母数字串。我们将仅使用单个的 Record 类的方法 char key_letter(int position)，它返回键中给定 position 的字符（如果键的长度小于 position 则返回一个空白）。就如 9.5 节，辅助函数 int alphabetic_order(char symbol)返回字符 symbol 的字母位置。根据我们先前的约定，这个函数对非空白、非字母的字符将返回一个值 27，对于空白字符返回一个值 0。在链式实现中，trie 包含了一个指向根的指针。

```
class Trie{
public:                    //  Add methcd prototypes here.
private:                   //  data members
  Trie_node*root;
};
```

trie 的每个结点需要存储指向 Record 的指针和指向其分支的指针数组，其分支对应于能由函数 position 返回的 28 种结果。这样我们得到下面的说明：

```
const int num_chars=28;
struct Trie_node {
//   data members
  Record*data;
  Trie_node*branch[num_chars];
//  constructors
  Trie_node();
};
```

Trie_node 的构造函数简单地将结点中所有的指针成员置为 NULL。

11.2.4 查找 Trie

查找过程成为下面的 Trie 方法：

```
Error_code Trie::trie_search(const Key &target,Record &x)const
/*Post: If the search is successful, a code of success is returned, and the output
        parameter x is set as a copy of the Trie's record that holds target.Otherwise,a
        code of not_present is returned.
  Uses:Methods of class Key.*/
```

```
{
  int position=0;
  char next_char;
  Trie_node*location=root;
  while(location!=NULL&&(next_char=target.key_letter(position))!=''){
                  //  Terminate search for a NULL location or a blank in the target.
   location=location->branch[alphabetic_order(next_char)];
                           //  Move down the appropriate branch of the trie.
   position++;
                           //  Move to the next character of the target.
  }
  if (location!=NULL&&location->data!=NULL){
     x=*(location->data);
     return success;
  }
  else
     return not_present;
}
```

构造 while 循环的终止条件以避免或者超出空的 trie 结点或者越过 Key 的末尾。在此循环的结束之处，location(如果不是 NULL)指向 trie 中对应于目标的结点。

11.2.5 Trie 的插入

将一个新的键加入到 trie 的过程非常类似于查找这个键：必须向下追踪 trie 到某个合适的位置并将 data 指针设置成指向新键的信息记录。如果在此过程中，遇到了 trie 中的 NULL 分支，则不必终止查找，而是必须创建新结点并将它们放到 trie 中，从而完成对应于新键的路径。因而我们得到下面的方法：

```
Error_code Trie::insert(const Record&new_entry)
/* Post: If the Key of new_entry is already in the Trie, a code of duplicate_error is
        returned. Otherwise, a code of success is returned and the Record new_entry is
        inserted into the Trie.
Uses: Methods of classes Record and Trie_node. */
{
  Error_code result=success;
  if(root==NULL) root=new Trie_node; //  Create a new empty Trie.
  int position=0;              //  indexes letters of new_entry
  char next_char;
  Trie_node*location=root; //  moves through the Trie
  while (location!=NULL &&
        (next_char=new_entry.key_letter(position))!=''){
   int next_position=alphabetic_order(next_char);
   if (location->branch[next_position]==NULL)
      location->branch[next_position]=new Trie_node;
   location=location->branch[next_position];
   position++;
  }
  //  At this point, we have tested for all nonblank characters of new_entry.
  if(location->data!=NULL) result=duplicate_error;
  else location->data=new Record(new_entry);
  return result;
}
```

11.2.6 Trie的删除

用于查找和插入的一般方案通常也适用于从trie中删除键。我们沿着对应于待删键的路径向下追踪，当到达合适的结点时，我们将相应的data成员设置为NULL。然而，如果现在这个结点的所有成员为NULL(所有的分支和data成员)，则应删除这个结点。为此，我们可以建立一个指针栈，其中的指针指向从根到最后到达结点的路径上的结点。作为另一选择，我们可以在删除算法中使用递归并避免采用外部栈。这两种方案都将留作练习。

11.2.7 Trie的评价

*** 与二叉查找的比较

查找trie(或插入到trie)所需要的步骤与组成键的字符数成比例，而不是像在其他基于树的查找中与键的数目的对数成比例。如果这个字符数相对于(底为2的)键的数目的对数较小，则证明trie优于二叉树。比如，如果键是由5个字母的所有可能的序列组成，则trie能在5个循环中查找$n=26^5=11\ 881\ 376$个键的任何一个，而二叉查找的最好情况是$\lg n\approx 23.5$次键比较。

然而，在许多应用中，键中的字符数比较多，且实际出现的键集在所有可能的键集当中是相对稀少的。在这些应用中，查找trie所要求的循环次数可能大大超过二叉查找所需要的键比较次数。

*** 词典书边标目

最后，最好的解决方案可能是组合这些方法。trie能用于键的前几个字符，然后对剩余的键采用其他方法。如果我们返回去讨论词典中书边标目的例子，可以看出，我们实际上使用了一个多路分支去查找词的第一个字母，但随后我们使用其他的查找方法从这些具有同样首字母的词中找出想要的词。

练习11.2

E1. 画出由如下每个键集构造的trie：

(1)仅仅包含1,2,3的三位整数(十进制表示)。

(2)由a,b,c,d组成的所有3个字母的序列，序列的第一个字母是a。

(3)所有四位二进制整数(由0和1构成)。

(4)由字母a,e,r构成的下列单词：

a ear re rare area are ere era rarer rear err

(5)由字母g,i,n构成的下列单词：

gig i inn gin in inning gigging ginning

(6)由字母a,b,l,p构成的下列单词：

pal lap a papa al papal all ball lab

E2. 编写方法遍历trie且以字母表次序打印出所有的单词。

E3. 编写方法遍历 trie 且打印出所有的单词，首先以单词的长度确定次序，较短的单词优先，其次对于长度相等的词则以字母表次序打印。

E4. 编写方法从 trie 中删除一个单词。

程序设计项目 11.2

P1. 为 trie 构造菜单驱动演示程序。键应是由 26 个小写字母构成的单词，长度不超过 8 个字符长。(除了键以外)保存在记录中的惟一信息是表示词何时插入的序列号。

11.3 外部查找:B-树

我们已假设在整本书中所有数据结构都保存在高速内存中，即我们仅考虑内部信息检索。这个假设对许多应用是合理的，但对许多其他的重要应用，可能需要查找和检索存储在文件中的记录。

11.3.1 访问时间

*** 存储块

从高速内存中访问和检索单词所要求的时间至多是微秒，而查找磁盘上一个特定记录所要求的时间却需要以毫秒度量，软盘甚至超过 1 秒。因此，就单次访问所要求的时间，外部检索比内部检索多上成百上千倍。另一方面，当记录位于磁盘上的时候，标准的实践不是仅仅读取一个单词，而是立刻读出一大页或一大块信息，块的典型大小从 256 到 1024 个字符或字。

在外部查找中我们的目标是必须使磁盘访问次数最少，因为每次访问对比于内部计算来说都花了非常长的时间。然而，每次外部访问，我们都得到一个存储块，这个块可以容纳若干条记录。我们能够利用这些记录对下一步要访问哪个块做出多路决策，因此，多路树特别适合外部查找。

11.3.2 多路查找树

二叉查找树可以直接推广到多路查找树，在多路查找树中，对某个称为树的阶(order)的整数 m，每个结点至多有 m 个孩子。k 是孩子数，如果 $k \leqslant m$，则结点恰恰只包含 $k-1$ 个键，它们将子树中的所有键分成 k 个子集。如果这些子集中有一些是空的，则树中对应的孩子也是空的。图 11.8 显示了 5-路查找树(每个结点中有 1 到 4 个元素)，其中某些结点的一些孩子是空的。

11.3.3 平衡多路树

我们的目标是设计简化文件访问的多路查找树，因此希望树的高度尽可能小。我们能

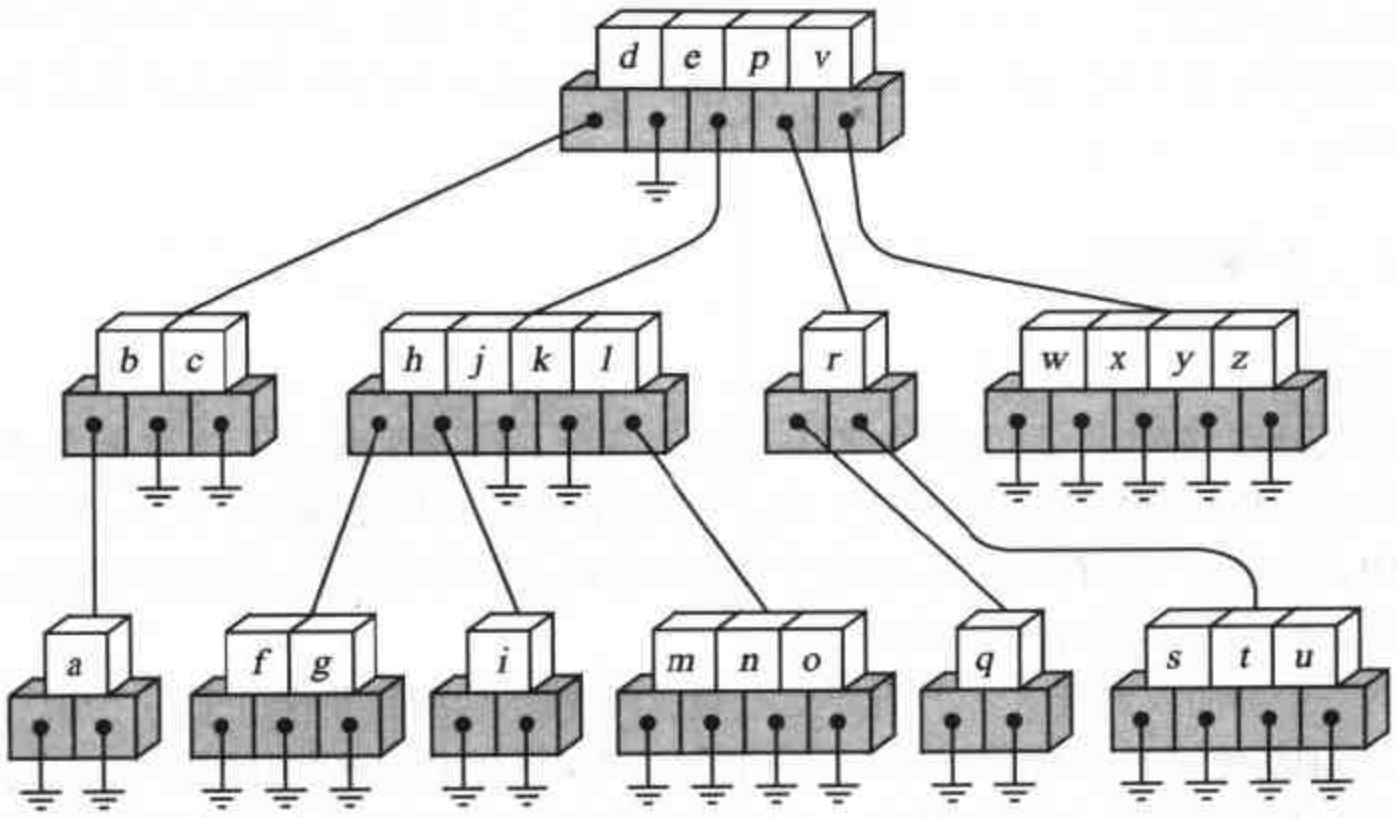


图 11.8 5-路查找树(不是 B-树)

通过以下要求实现这个目标:首先,树叶以上没有空子树出现(所以将键划分成子集是最高效的);其次,所有的树叶都在同一层(所以能保证所有查找终止时都进行了大约同样次数的访问);第三,每个结点(除树叶外)至少有某个较小数目的孩子,我们将要求每个结点(除树叶)至少有最多可能数目的一半的孩子。这些条件将产生下面的形式化定义:

定义 阶为 m 的 B-树是一棵 m-路树,其中:

1. 所有的树叶在同一层。
2. 除了根以外的所有内部结点至多有 m 个非空孩子,至少有$\lceil m/2 \rceil$个非空的孩子。
3. 每个内部结点中的键数都比它的非空孩子数少 1,而且这些键以查找树的方式划分孩子中的键。
4. 根至多有 m 个孩子,但如果它不是树叶则只有 2 个孩子,或者如果树仅由根本身组成,则它没有孩子。

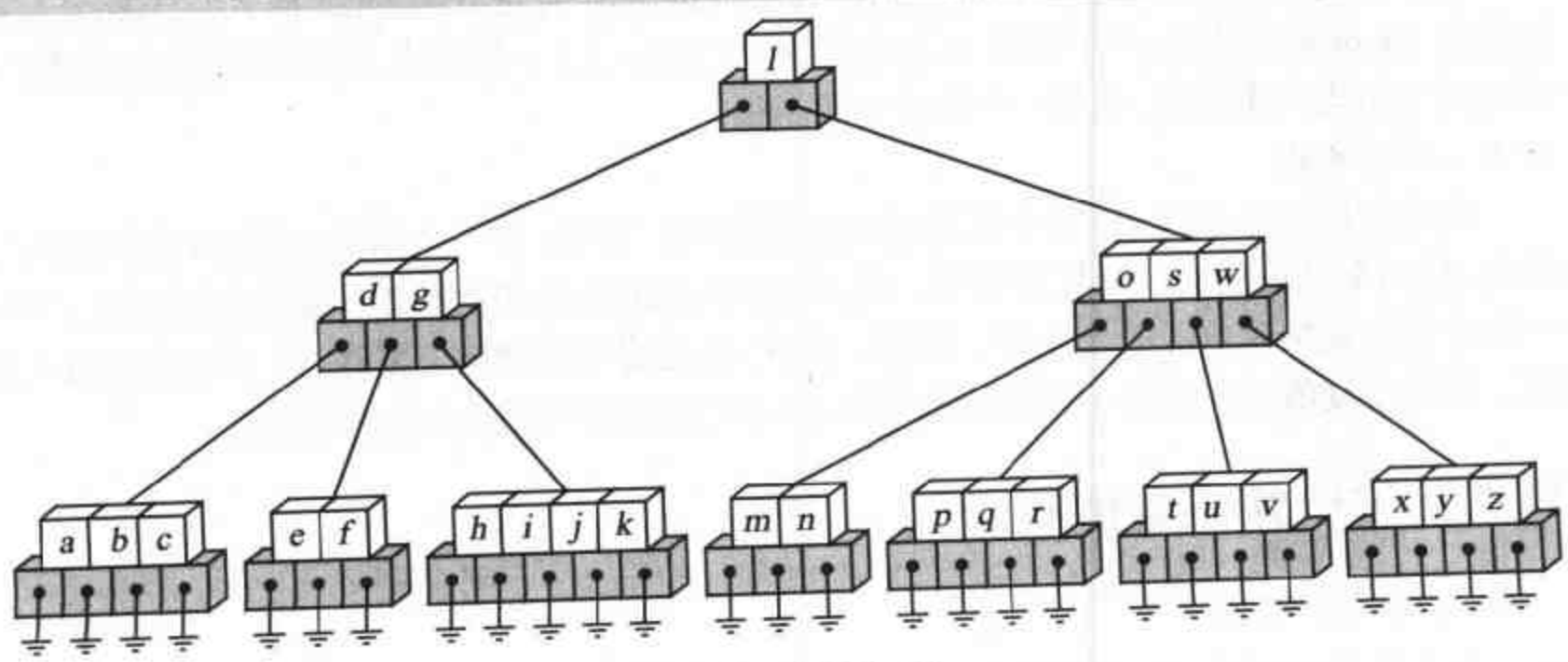


图 11.9 5 阶 B-树

图 11.8 中的树不是 -棵 B-树，因为一些结点有空的孩子，另一些结点有太少的孩子，而且树叶也不是都在同一层。图 11.9(见上页)显示了阶为 5 的 B-树，它的键是字母表中的 26 个字母。

11.3.4 B-树的插入

*** 方法

所有的结点都在同一层的条件使 B-树具有一个典型的性能：与二叉树相比，B-树不允许在树叶上生长，而要求它们在根上生长。因此插入的一般方法如下：首先，查找新的键是否在树中，(如果这个键真的是新的)此查找将在树叶上以失败告终，然后将新键加入到叶结点，如果这个结点先前没有满，则插入完成。

当将一个键加入到满结点，则此结点分裂成两个结点，而且并排位于同一层。除了中间键以外，其他的键都放入到两个新结点中的某一个，而将中间键上升到树中的父结点处插入。后来的查找就在树上进行，因此，与中间键进行比较能指示进入合适的子树查找。当将键加入到一个已满的根时，根分裂成两个结点，中间的键上升并成为一个新根。这是 B-树惟一的一次高度增长。

通过研究如图 11.10 所示的 5 阶 B-树的生长，用这样的例子可以很好地解释清楚这个过程。我们将以给定的次序将下面这些键插入到初始为空的树中：

a g f b k d h m j e s i r x c l n t u p

*** 结点分裂

最前面的 4 个键将插入到一个结点中，如图 11.10 的第一个图所示。当这些键被插入时，它们被排成合适的次序。然而对第 5 键 *k*，结点中已没有空间，所以 *k* 的插入将导致第一个结点分裂成两个结点，中间的键 *f*，上移进入一个新的结点，这个结点是一个新的根。因为分裂过的结点现在只是半满，因此接下来的 3 个键能毫无困难地插入。然而注意这些简单插入会要求结点内的键进行重排。当下一次插入 *j* 时，再次分裂一个结点，而且这次 *j* 本身就是中间键，因而上移加入根中的 *f*。

*** 向上繁殖

接下来的几次插入过程与前面类似。在最后插入 *p* 时更有趣：此次插入首先要分裂最初包含 *k l m n* 的结点，将中间键 *m* 向上发送到包含 *c f j r* 的结点，然而该结点也是满的，因此这个结点再次分裂，创建一个包含 *j* 的新根。

*** 改善平衡

此处给出两个关于 B-树生长的注解是适宜的。首先，当一个结点分裂时，它产生两个结点，这两个结点现在仅仅是半满的。因而，以后的插入更可能不必分裂结点而完成，因此一次分裂为几次简单的插入铺平了道路。其次，总是将中间键向上发送，而不必是刚插入的键。因此重复的插入有利于改善树的平衡，而不用管键碰巧以什么次序到达。

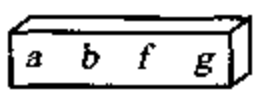
11.3.5 C++算法：查找和插入

*** 指针和磁盘访问

为了用 C++开发 B-树的查找和插入算法，我们从建立 B-树所需的定义开始。为简化

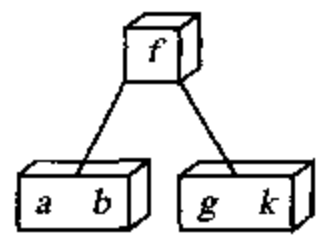
1.

a, g, f, b:



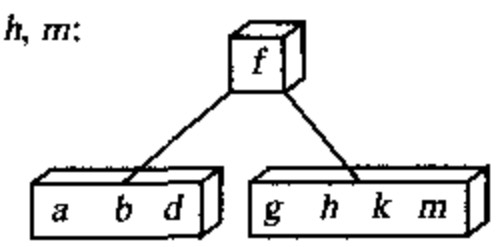
2.

k:



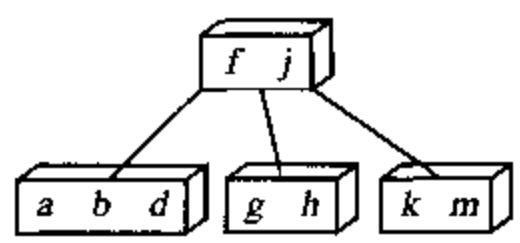
3.

d, h, m:



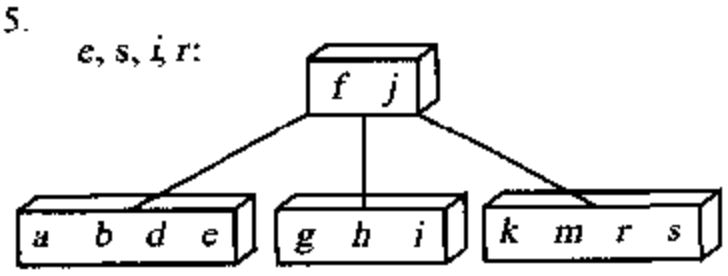
4.

j:



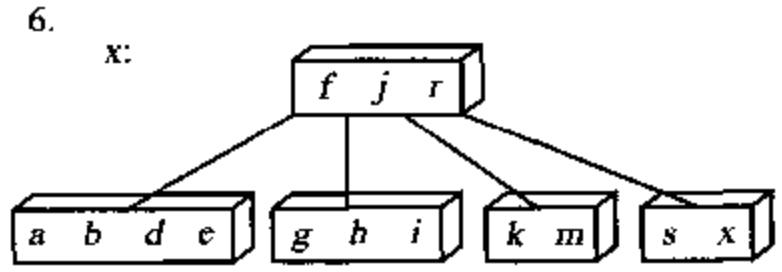
5.

e, s, i, r:



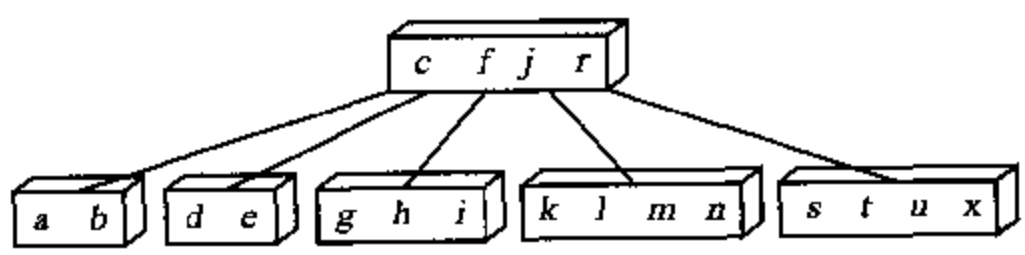
6.

x:



7.

c, l, n, t, u:



8.

p:

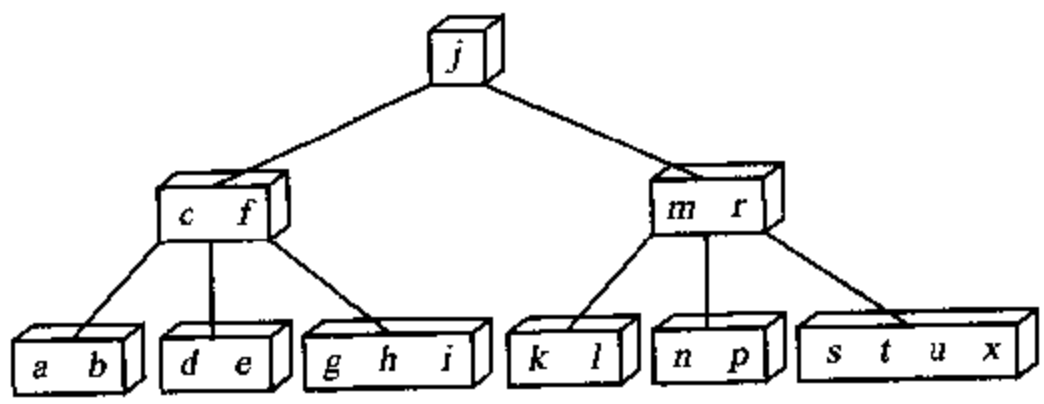


图 11.10 B-树的生长

起见，我们将完全在高速内存中构造 B-树，使用指针去描述它的结构。在大多数应用中，这些指针将被磁盘上各种块或页的地址所代替，而且对指针的引用将变成磁盘访问。

1. 声明

* * * 参数：阶

我们让客户程序自由选择希望存储在 B-树中的记录。相应地，我们的 B-树类和相应

的结点类，将是以 Record 类作参数的模板。我们也将加入另一个模板参数，它是表示 B-树的阶的一个整数。它允许客户程序用如下简单的定义对 B-树对象进行自定义：B_tree<int, 5>sample_tree，它把 sample_tree 定义为一个保存整数记录的 5 阶 B_tree。我们得到下面的类模板说明：

```
template<class Record,int order>
class B_tree {
public:                          //  Add public methods.
private:                         //  data members
  B_node<Record,order>*root;
                                 //  Add private auxiliary functions here.
};
```

* * * 顺序表

在 B-树的每个结点中，我们需要一个元素表和一个指向结点孩子的指针表。由于这些表是短整型的，为简单起见，我们将使用顺序数组和单独的数据成员 count 表示它们。

```
template<class Record, int order>
struct B_node {
//  data members:
  int count;
  Record data[order-1];
  B_node<Record,order>*branch[order];
//  constructor:
  B_node();
};
```

* * * data 和 branch 的下标的意义

数据成员 count 给出了 B_node 中的记录数。如果 count 非零，则该结点有 count+1 个孩子。branch[0]指向包含键小于 data[0]中的键的所有记录的子树，对 position 在 1～count-1 (包括 1 和 count-1)之间的每个值，branch[position]指向键严格位于 data[position-1]和 data[position]的之间的子树，且 branch[count]指向键大于 data[count-1]的键的子树。

B_node 构造函数创建一个空结点，它在新建立的结点中将 count 的值设置成 0 而使它为空。

2. 查找

作为第一个简单的例子，我们编写一个方法去查找 B-树中与目标记录的键相匹配的记录。在我们的查找方法中，我们将像通常一样假设记录能用标准的运算符进行比较。就像二叉树中的查找一样，我们从调用一个递归辅助函数开始。

```
template<class Record, int order>
Error_code B_tree<Record, order>::search_tree(Record &target)
/*Post: If there is an entry in the B-tree whose key matches that in target, the parameter
        target is replaced by the corresponding Record from the B-tree and a code of
        success is returned. Otherwise a code of not present is returned.
  Uses: recursive_search_tree*/
{
```

```
  return recursive_search_tree(root,target);
}
```

* * * 参数

辅助函数 recursive_search_tree 的输入参数是一个指向 B-树子树根的指针和保存目标键的记录。此函数返回一个 Error_code 去报告 target 和 B_tree 的元素是否匹配:如果成功则更新参数 target 的值去匹配 B-树中找到的记录。

* * * 查找结点

通过对树向下进行查找的通用方法类似于对二叉查找树进行查找。然而,在多路树中,必须更全面地检查每个结点以找到下一步要采用的分支,我们用另一个辅助的 B-树函数 search_node 来实施这个检查,它查找存储在 current 结点中的记录。函数 search_node 使用了一个输出参数 position,如果在 current 结点中找到目标,则 position 是该目标的下标,否则是要继续查找的分支的下标。

```
template<class Record,int order>
Error_code B_tree<Record,order>::recursive_search_tree(
          B_node<Record,order>*current,Record &target)
/*Pre: current is either NULL or points to a subtree of the B_tree.
  Post: If the Key of target is not in the subtree,a code of not_present is returned.
       Otherwise,a code of success is returned and target is set to the corresponding
       Record of the subtree.
  Uses: recursive_search_tree recursively and search_node*/
{
  Error_code result=not_present;
  int position;
  if (current!=NULL){
     result=search_node(current,target,position);
     if(result==not_present)
        result=recursive_search_tree(current->branch[position],target);
     else
        target=current->data[position];
  }
  return result;
}
```

* * * 尾部递归

这个函数采用递归编写,它的结构展现了与将要设计的插入函数的结构之间的相似性。然而,这个递归是尾部递归,因此如果需要的话,可以简单地用迭代去替换它。

3. 查找结点

这个函数必须确定目标是否在当前结点中存在,如果不存在的话,它将找出 count+1 个分支中的哪一个将包含目标键。我们将计数器 position 初始化为 0 并一直递增它,直至到达目标或者超过目标。

```
template<class Record,int order>
Error_code B_tree<Record,order>::search_node(
  B_node<Record,order>*current,const Record &target,int&position)
/*Pre: current points to a node of a B_tree.
  Post: If the Key of target is found in *current,then a code of success is returned,the
```

```
        parameter position is set to the index of target, and the corresponding
        Record is copied to target. Otherwise, a code of not_present is returned, and
        position is set to the branch index on which to continue the search.
  Uses: Methods of class Record.*/
{
  position=0;
  while(position<current->count && target>current->data[position])
    position++;          //    Perform a sequential search through the keys.
  if(position<current->count&&target==current->data[position])
    return success;
  else
    return not_present;
}
```

***二叉查找

对于大阶的B-树，应对这个函数加以修改，使用二叉查找而不是顺序查找。在一些应用中，大量的信息存储在B-树的每个记录中，因此B-树的阶将相对较小，从而一个结点内的顺序查找是合适的。在许多应用中，结点中仅保存键，所以阶很大，而且二叉查找要用于定位结点中的键的位置。

***二叉查找树

然而，另一种可能性是使用链式二叉查找树代替每个结点元素的顺序数组，我们将在这一章的后而详细地研究这种可能性。

4. 插入：主函数

B-树的插入能最自然地采用递归函数表示，因为在子树中的插入完成后，而可能剩下的(中间的)插入必须再次在树的更高层中进行。递归允许我们跟踪树内的位置，并在树中一路向上倒退，不需要显式的辅助栈。

***参数

照常，我们要求待插入的键没有在树中出现，于是插入方法只要一个参数new_entry，它是待插入的记录。然而对递归来说，我们需要3个附加的输出参数。第1个是current，它是当前正被处理的子树的根。如果*current需要分裂以容纳new_entry，则递归函数将返回代码overflow(因为对于键来说已经没有空位)，并确定将要再次插入到树中更高层的(中间)记录。当这种情况发生时，我们将采用如下约定：原来的结点*current包含左边一半元素，新的结点包含右边一半元素。当出现这样的分裂时，第2个输出参数median给出中间的记录，而第3个输出参数right_branch给出指向新结点的指针，这个结点是该子树先前的根*current的右边一半。

为了使所有这些参数简单易懂，我们将在称为push_down的函数中进行递归。图11.11说明了结点分裂时的这种情况。

此递归以B_tree的方法insert开始。如果对函数push_down的最外层调用返回的代码为overflow，则仍有一个记录median要重新插入到B-树中。然后我们必须创建一个新根去保存这个记录，而且整棵B-树的高度将因此而增加。这是B-树高度增长的惟一方法。

插入方法显示如下：

```
template<class Record, int order>
```

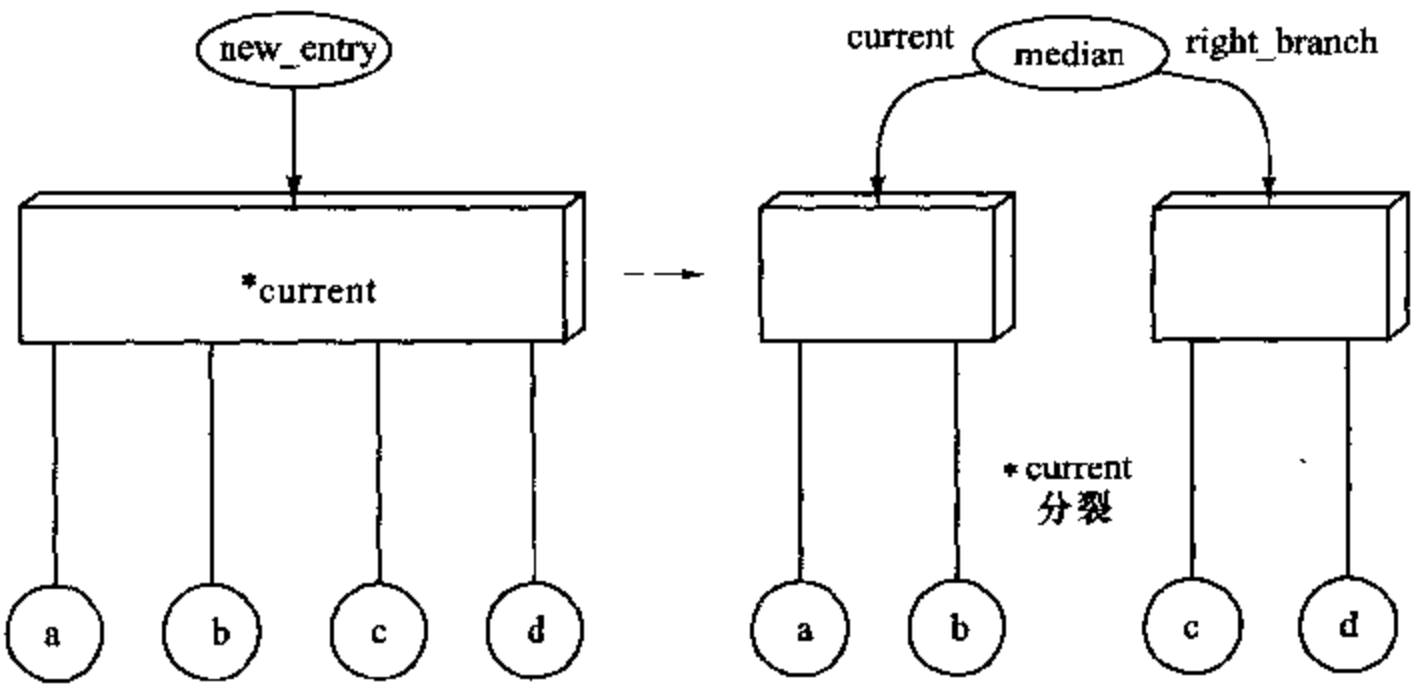


图 11.11 结点分裂时 push_down 函数的操作

```
Error_code B_tree<Record,order>::insert(const Record &new_entry)
/*Post: If the Key of new_entry is already in the B_tree, a code of duplicate_error is
        returned. Otherwise, a code of success is returned and the Record new_entry is
        inserted into the B-tree in such a way that the properties of a B-tree are
        preserved.
  Uses: Methods of struct B_node and the auxiliary function push_down.*/
{
  Record median;
  B_node<Record,order>*right_branch,*new_root;
  Error_code result=push_down(root,new_entry,median,right_branch);
  if (result==overflow){     //   The whole tree grows in height.
                             //   Make a brand new root for the whole B-tree.
    new_root=new B_node<Record,order>;
    new_root->count=1;
    new_root->data[0]=median;
    new_root->branch[0]=root;
    new_root->branch[1]=right_branch;
    root=new_root;
    result=success;
  }
  return result;
}
```

5. 子树的递归插入

*** 停止规则

下面我们介绍递归函数 push_down，它使用参数 current 指向待查找的子树的根。在 B-树中，新的记录首先是插入到树叶，因此我们使用条件 current==NULL 去终止递归，也就是说，我们会继续在树中向下移动去查找 new_entry，直到遇到一棵空树。由于 B-树不是通过增加新的树叶生长，因而我们不是立即插入 new_entry，而是返回一个 overflow 代码（因为空子树不能有插入的记录），并将记录向上送回（现在称为 median）以用于以后的插入。

*** 重新插入记录

当递归调用返回一个 overflow 代码时，记录 median 还没被插入，我们试图将它插入到

当前结点中。如果当前结点还有空间的话，则完成插入。否则，结点 * current 将分裂成 * current 和 * right_branch，并且将(可能不同的)中间记录向上送到树上。函数使用了 3 个辅助函数：search_node，push_in 和 split，search_node 对查找操作作用相同；push_in 在有空间时将中间记录放入结点 * current；split 将一个已满的结点 * curren 分裂成两个结点，这两个结点将是 B-树同一层的兄弟结点。

```
template<class Record, int order>
Error_code B_tree <Record, order> :: push_down (
                    B_node<Record, order> *current,
                    const Record &new_entry,
                    Record &median,
                    B_node<Record, order> *&right_branch)
/* Pre: current is either NULL or points to a node of a B_tree.
  Post: If an entry with a Key matching that of new_entry is in the subtree to which
        current points, a code of duplicate_error is returned. Otherwise, new_entry is
        inserted into the subtree: If this causes the height of the subtree to grow, a
        code of overflow is returned, and the Record median is extracted to be reinserted
        higher in the B-tree, together with the subtree right_branch on its right. If the
        height does not grow, a code ofsuccess is returned.
  Uses: Functions push_down (called recursively), search_node, split_node, and push_
        in. */
{
  Error_code result;
  int position;
  if (current==NULL) {
            //  Since we cannot insert in an empty tree, the recursion terminates.
    median=new_entry;
    right_branch=NULL;
    result=overflow;
  }
  else {                              //   Search the current node.
    if (search_node(current, new_entry, position) ==success)
      result=duplicate_error;
    else {
      Record extra_entry;
      B_node<Record, order> *extra_branch;
      result=push_down (current->branch[position], new_entry,
                        extra_entry, extra_branch);
      if(result==overflow) {
                              //   Record extra_entry now must be added to current
        if (current->count<order- 1) {
          result=success;
          push_in(current, extra_entry, extra_branch, position);
        }
        else split_node(current, extra_entry, extra_branch, position,
                        right_branch, median);
        //   Record median and its right_branch will go up to a higher node.
      }
    }
  }
  return result;
}
```

6. 将键插入到结点中

下一个辅助函数是 push_in，只要有用于插入的空间，则该函数将 Record 元素及其右侧指针 right_branch 插入到结点 * current。在图 11.12 中说明了这种情形。

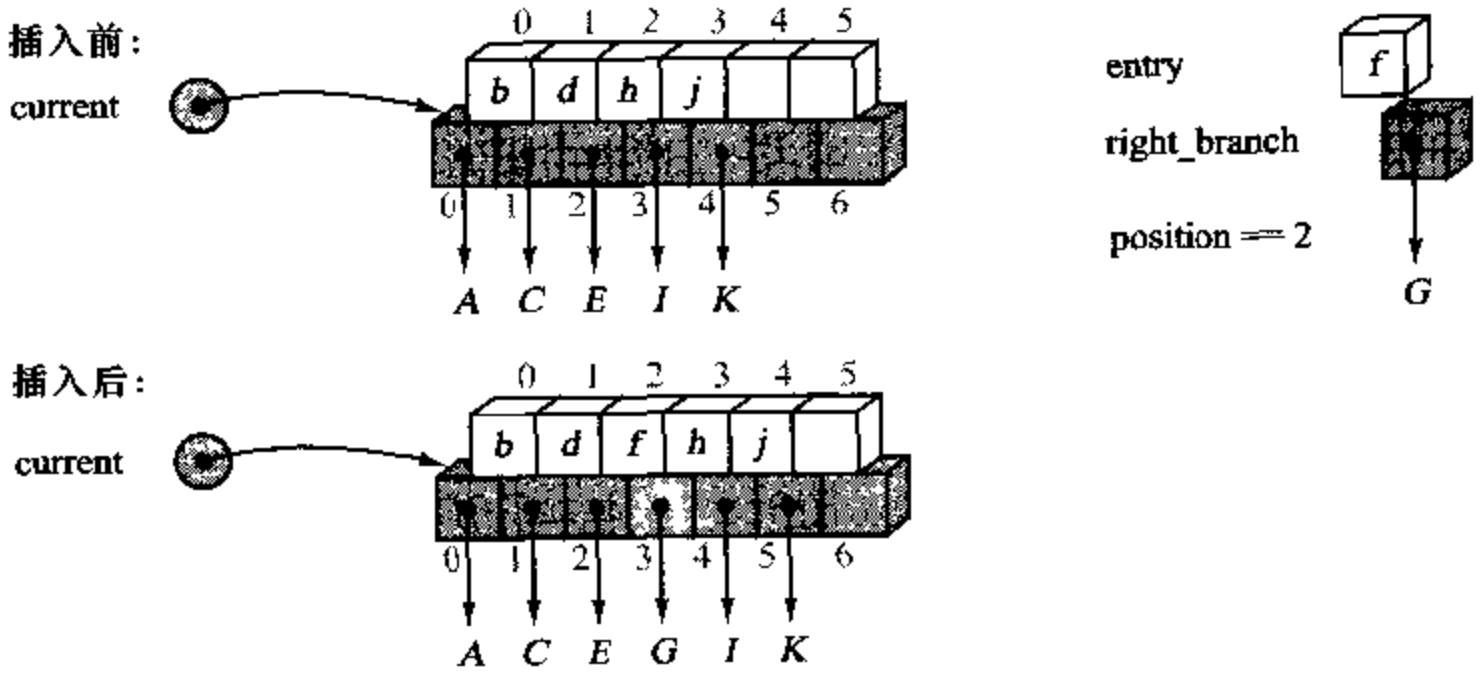


图 11.12　push_in 函数的操作

```
template<class Record, int order>
void B_tree<Record, order> :: push_in(B_node<Record, order>*current,
  const Record &entry, B_node<Record, order>*right_branch, int position)
/*Pre: current points to a node of a B_tree. The node *current is not full andentry
       belongs in*current at index position.
  Post:entry has been inserted along with its right-hand branch right_branch into*
       current at index position.*/
{
  for(int i=current->count; i> position; i--){
                                //   Shift all later data to the right.
    current->data[i]=current->data[i-1];
    current->branch[i+1]=current->branch[i];
  }
  current->data[position]=entry;
  current->branch[position+1]=right_branch;
  current->count++;
}
```

7. 分裂满结点

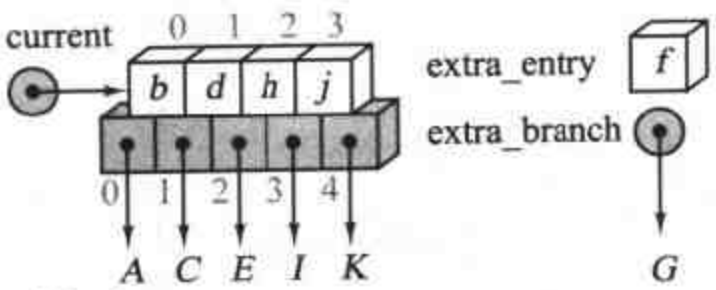
* * * 大致概要

最后的辅助插入函数 split_node，必须将一个具有子树指针 extra_branch 的记录 extra_entry 插入到一个满结点 * current 中，而且分裂出右边一半作为一个新结点 * right_half。该函数也必须从它的结点中删除中间记录并将它向上发送用于随后的重新插入。

当然，直接将记录 extra_entry 插入满结点是不可能的：我们必须先确定 extra_entry 是属于左半部还是右半部，相应地分开结点，然后再将 extra_entry 插入到合适的那一半。当

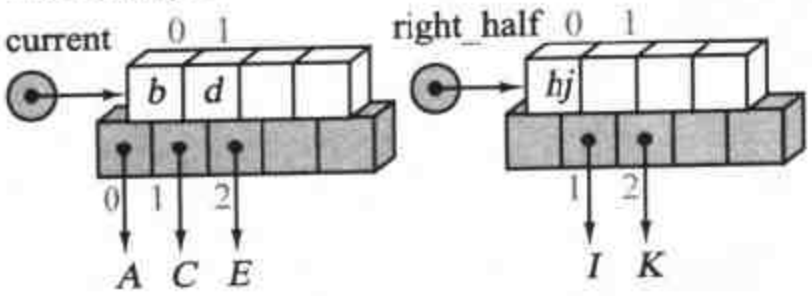
所有这些工作进行期间，我们将划分这个结点，使得 Record median 是左边那一半中最大的元素。在图 11.13 中说明了这种情形。

情形一：position == 2; order == 5;
(extra_entry 属于左半部分)

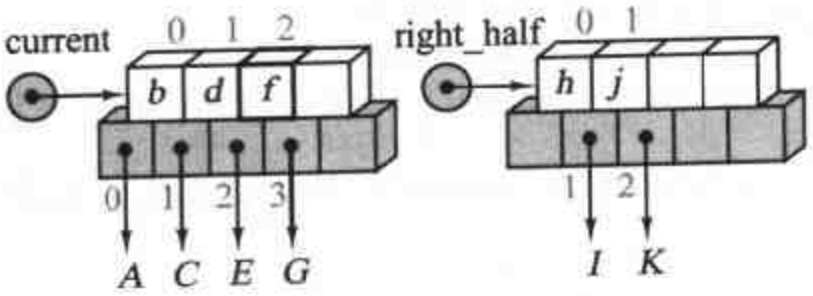


mid = 2;

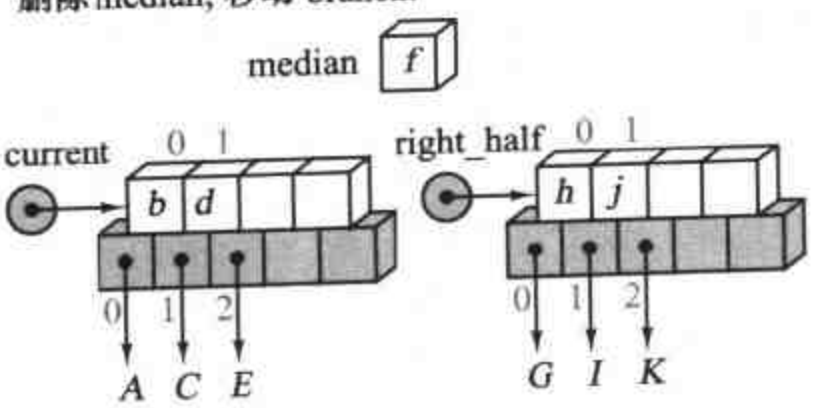
将元素右移:



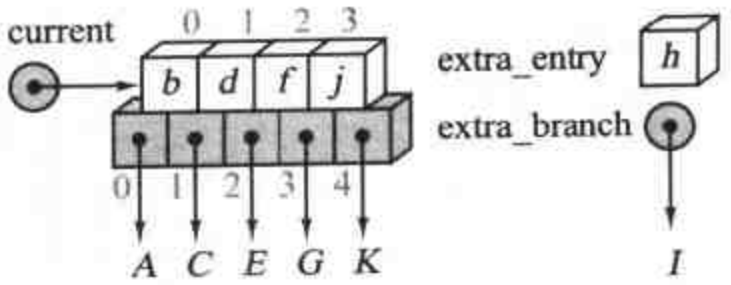
插入extra_entry和 extra_branch:



删除median; 移动 branch:

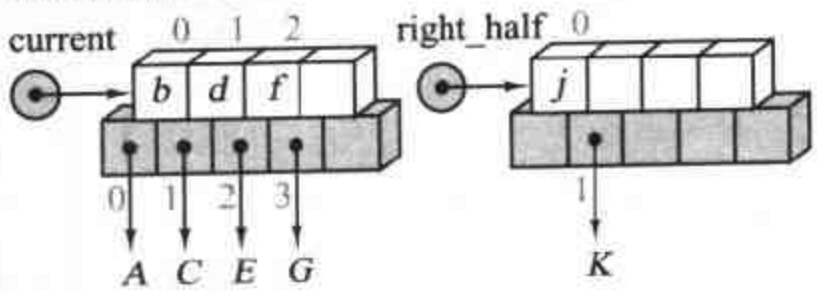


情形二：position == 3; order == 5;
(extra_entry 属于右半部分)

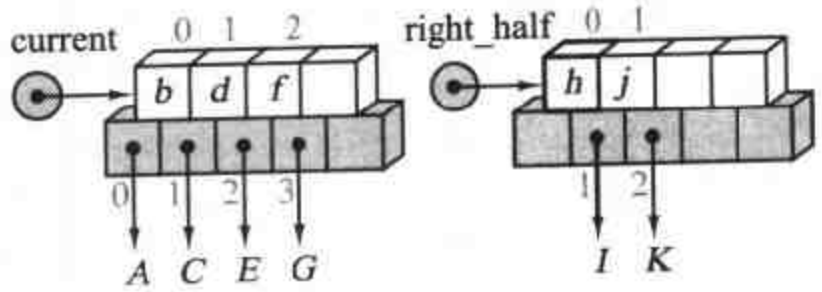


mid = 3;

将元素右移:



插入extra_entry和 extra_branch:



删除median; 移动 branch:

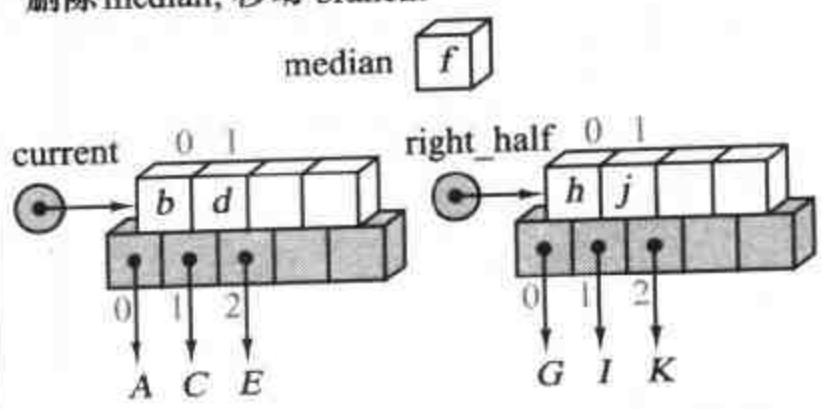


图 11.13 split 函数的操作

```
template<class Record, int order>
void B_tree<Record, order>::split_node(
  B_node<Record, order>*current, //   node to be split
  const Record &extra_entry,//   new entry to insert
  B_node<Record, order>*extra_branch, //   subtree on right of extra_entry
  int position,                  //   index in node where extra_entry goes
```

```
  B_node<Record,order>*&right_half,//   new node for right half of entries
  Record &median)                  //   median entry(in neither half)
/*Pre: current points to a node of a B_tree. The node*current is full,but if there were
       room,the record extra_entry with its right-hand pointer extra_branch would
       belong in*current at position position,0<=position<order.
  Post: The node *current with extra_entry and pointer extra_branch at position
       position are divided into nodes*current and *right_half separated by a Record
       median.
  Uses: Methods of struct B_node, function push_in.*/
{
  right_half=new B_node<Record,order>;
  int mid=order/2;                 //   The entries from mid on will go to right_half.
  if(position<=mid){               //   First case:extra_entry belongs in left_half.
    for(int i=mid; i<order-1; i++){//   Move entries to right_half.
      right_half->data[i-mid]=current->data[i];
      right_half->branch[i+1-mid]=current->branch[i+1];
    }
    current->count=mid;
    right_half->count=order-1-mid;
    push_in(current,extra_entry,extra_branch,position);
  }
  else {                           //   Second case: extra_entry belongs in right half.
    mid++;                         //   Temporarily leave the median in left half.
    for(int i=mid; i<order-1; i++){ //   Move entries to right_half.
      right_half->data[i-mid]=current->data[i];
      right_half->branch[i+1-mid]=current->branch[i+1];
    }
    current->count=mid;
    right_half->count=order-1-mid;
    push_in(right_half,extra_entry,extra_branch,position-mid);
  }
    median=current->data[current->count-1];
                                   //   Remove median from left half.
    right_half->branch[0]=current->branch[current->count];
    current->count--;
}
```

11.3.6 从B-树删除

1. 方法

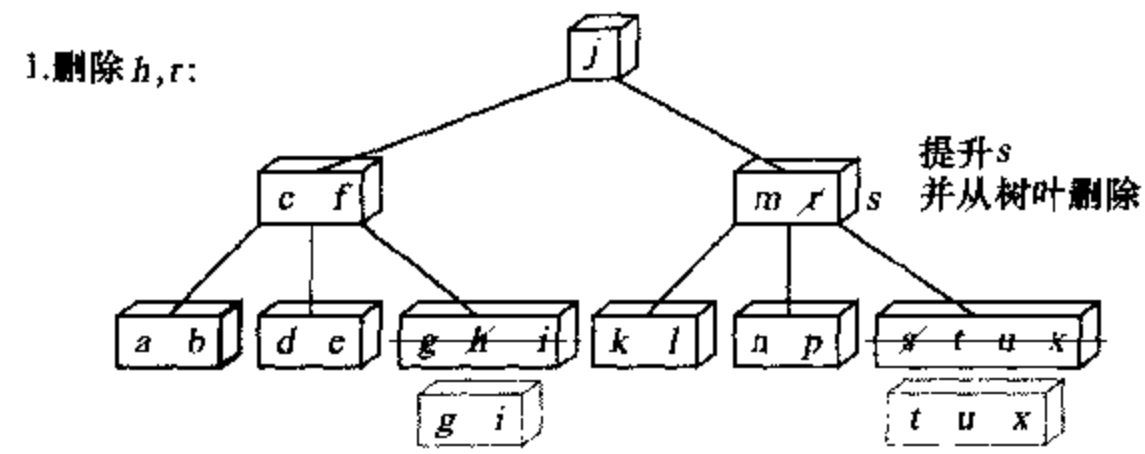
在插入期间，新元素总是先进入树叶。对于删除，我们也希望从树叶处删除元素。如果要删除的元素不在叶结点内，则可以保证它在自然键序下的直接前趋(或后继)是在树叶里(证明它!)。因此我们能将此直接前趋(或后继)提升到被删元素占据的位置中，然后从叶结点删除元素。

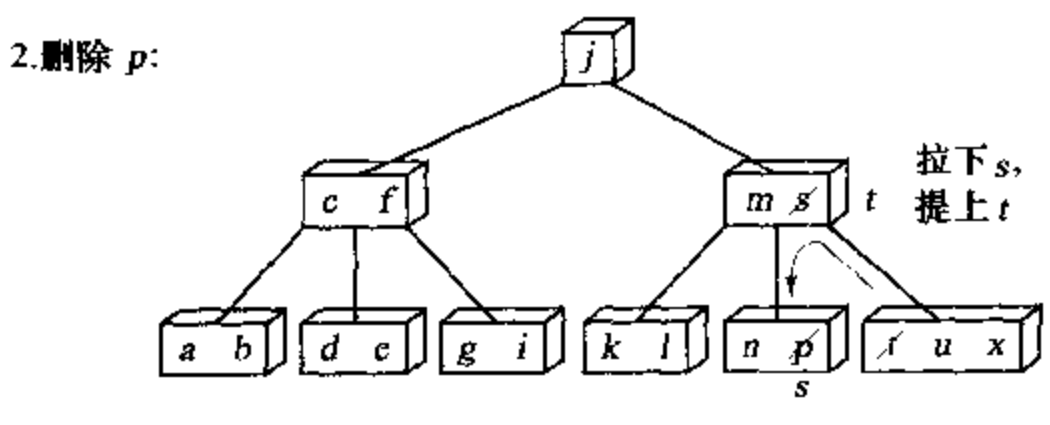
如果树叶包含多于最少数目的元素，则无需更多操作就能删除其中一个元素。如果树叶包含最少数目，则我们首先查看两个树叶(或者当一个结点在外部的情况下，是一个树叶)，它们直接互相毗邻，并且是同一个结点的孩子。如果其中的一个结点有超过最少数目的元素，则能够将这些元素中的某一个移入到双亲结点，而将双亲结点的元素移到发生删除的树叶。最后，如果毗邻的树叶仅有最少数目的元素，则这两个树叶和来自双亲结点的中间元素能一起合并为一个新的树叶，它将包含所允许的最大数目的元素。如果这一步导致双

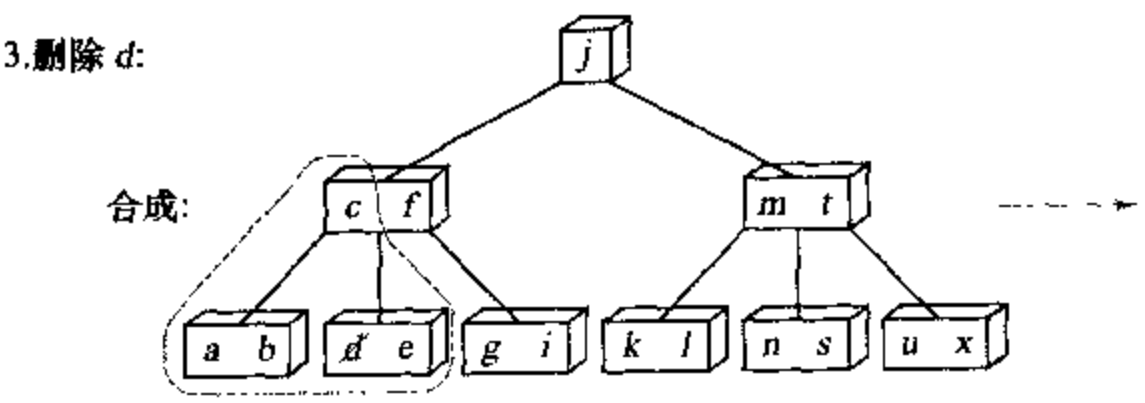
亲结点具有太少元素，则这个过程将向上传播。在有限的情况下，需要从根中删除最后的元素，然后减少树的高度。

2. 例子

图 11.14 显示了我们前面介绍的 5 阶 B-树的删除过程。首先是从一个树叶中删除 h，此树叶含有的元素数超过最小值，因此不会引起问题。其次删除 r，这次不是从树叶中删除，







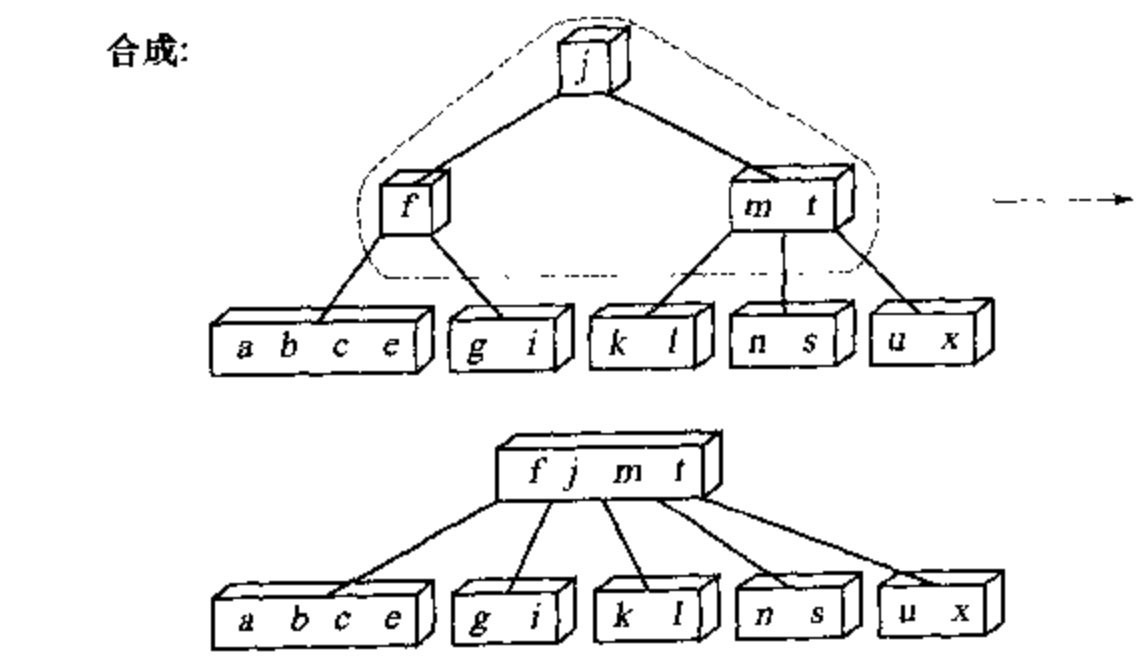


图 11.14　从 B-树删除

因此 r 的直接后继 s 被提升到 r 的位置，然后将 s 从它的树叶删除。最后删除 p，使它的结点剩下太少的元素，所以来自其双亲结点的键 s 因此下降，它的位置被键 t 代替。

* * * 合并结点

d 的删除有更广泛的结果，这个删除使得此结点有太少的元素，而且它的兄弟结点中含有的元素不多于一个。因此该结点被合并到一个兄弟中，并且具有来自双亲结点的中间元素，如第一个图中的点线和第二个图中的合成结点 $a\ b\ c\ e$ 所示。然而，这个过程使双亲结点只剩下一个键 f，因此树的最顶部的 3 个结点必须合并，从而产生图 11.14 中最后一个图所示的树。

3. C++实现

我们能用类似于插入算法使用的总体结构去编写删除算法。像往常一样，我们将使用递归，并用一个单独的方法启动递归。在内部递归调用期间，即使根结点有太少的元素，我们也允许递归函数返回而不是试图从双亲结点取下一个元素。外部调用要能够检测到这种情况的发生并按要求移动元素。当从根中删除最后的元素时，此空结点被删除，并且 B-树的高度收缩。

这个方法的实现是：

```
template<class Record, int order>
Error_code B_tree<Record, order>::remove(const Record &target)
/*Post: If a Record with Key matching that of target belongs to the B_tree, a code of
      success is returned and the corresponding node is removed from the B-tree.
      Otherwise, a code of not_present is returned.
  Uses: Function recursive_remove*/
{
  Error_code result;
  result=recursive_remove(root, target);
  if(root!=NULL && root->count==0){ //   root is now empty.
    B_node<Record, order>*old_root=root;
    root=root->branch[0];
    delete old_root;
  }
  return result;
}
```

4. 递归删除

大多数的工作是在递归函数 recursive_remove 中完成。它首先查找当前结点是否存在目标，如果找到 target 且当前结点不是树叶，则查找 target 的直接后继结点并以它代替当前结点，然后再将此后继结点删除。在树叶处删除比较简单，否则将通过递归继续此过程。当递归调用返回时，函数检查在合适的结点中是否仍有足够多的元素，如果没有，则它将按要求移动元素。这些步骤中有几处都用到了辅助函数。

```
template<class Record, int order>
Error_code B_tree<Record, order>::recursive_remove(
  B_node<Record, order>*current, const Record &target)
/*Pre: current is either NULL or points to the root node of a subtree of a B_tree.
  Post: If a Record with Key matching that of target belongs to the subtree, a code of
```

```
    success is returned and the corresponding node is removed from the subtree so
    that the properties of a B-tree are maintained. Otherwise, a code of not_present
    is returned.
  Uses: Functions search_node, copy_in_predecessor, recursive_remove (recursively),
    remove_data, and restore.*/
{
  Error_code result;
  int position;
  if(current==NULL) result=not_present;
  else {
    if(search_node(current,target,position)==success){
                              //    The target is in the current node.
      result=success;
      if (current->branch[position]!=NULL){ //  not at a leaf node
        copy_in_predecessor(current,position);
        recursive_remove (current->branch[position],
                          current->data[position]);
      }
      else remove_data(current,position); //  Remove from a leaf node.

    }
    else result=recursive_remove(current->branch[position],target);
    if(current->branch[position]!=NULL)
      if (current->branch[position]->count<(order-1)/2)
        restore(current,position);
  }
  return result;
}
```

5. 辅助函数

我们现在通过编写各种目的所需要的几个辅助函数来总结 B-树的删除过程。函数 remove_data 直接删除 B-树的结点元素及其右侧分支。这个函数仅仅当元素从树的树叶删除时被调用。

```
template<class Record, int order>
void B_tree<Record, order>::remove_data(B_node<Record, order>*current,
                                        int position)
/*Pre: current points to a leaf node in a B tree with an entry at position.
  Post: This entry is removed from*current.*/
{
  for (int i=position; i<current->count-1; i++)
      current->data[i]=current->data[i+1];
  current->count--;
}
```

当必须从非叶结点删除元素时，我们调用函数 copy_in_predecessor。在这种情况下，通过首先取元素左边的分支，然后再取最右边的分支，直至到达树叶，我们可以找到它(按键的次序)的直接前趋。然后用这个树叶中最右边的元素代替被删除的元素。

```
template<class Record, int order>
void B_tree< Record, order> <::copy_in_predecessor(
                    B node< Record, order>* current, int position)
```

```
/*Pre: current points to a non leaf node in a B-tree with an entry at position .
  Post: This entry is replaced by its immediate predecessor under order of keys.*/
{
  B_node<Record,order>*leaf=current->branch[position];
                          //   First go left from the current entry.
  while(leaf->branch[leaf->count]!=NULL)
    leaf=leaf->branch[leaf->count]; //   Move as far rightward as possible.
  current->data[position]=leaf->data[leaf->count-1];
}
```

最后，必须说明当递归调用将 root->branch[position]的计数减到低于最少元素数时，如何将它恢复到所要求的最少元素数。我们编写的函数有点偏向左边，也就是说，它看起来首先是到左边的兄弟去取元素，仅当左边没有多余的元素时才使用右边的兄弟。图 11.15 说明了所需要的步骤。

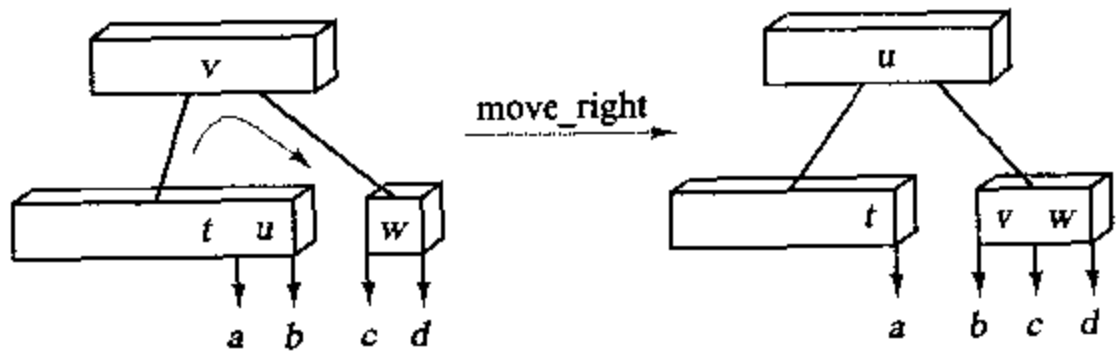


图 11.15 最少元素数的恢复

```
template<class Record,int order>
void B_tree<Record,order>::restore(B_node<Record,order>*current,
                                   int position)
/*Pre:  current points to a non-leaf node in a B-tree; the node to which current->branch
        [position] points has one too few entries.
  Post:An entry is taken from elsewhere to restore the minimum number of entries in the
       node to which current->branch[position] points.
  Uses: move_left ,move_right,combine.*/
{
  if(position==current->count) //   case:rightmost branch
    if(current->branch[position-1]->count> (order-1)/2)
       move_right(current,position-1);
    else
    combine(current,position);
  else if(position==0)     //   case:leftmost branch
    if(current->branch[1]->count> (order-1)/2)
       move_left(current,1);
    else
```

```
      combine(current,1);
  else                          //   remaining cases: intermediate branches
    if(current->branch[position-1]->count> (order-1)/2)
       move_right(current,position-1);
    else if(current->branch[position+1]->count> (order-1)/2)
      move_left(current,position+1);
    else
      combine(current,position);
}
```

从图 11.15 可以清楚地看出剩余 3 个函数 move_left,move_right 和 combine 的操作。

```
template<class Record,int order>
void B_tree<Record,order>::move_left(B_node< Record,order>* current,
                                     int position)
/*Pre: current points to a node in a B-tree with more than the minimum number of entries
       in branch position and one too few entries in branch position-1.
  Post: The leftmost entry from branch position has moved into current, which has now
       an entry into the branch position-1.*/
{
  B_node< Record,order>*left_branch=current->branch[position-1],
                       *right_branch=current->branch[position];
  left_branch->data[left_branch->count]=current->data[position-1];
                       //   Take entry from the parent.
  left_branch->branch[++left_branch->count]=right_branch->branch[0];
  current->data[position-1]=right_branch->data[0];
                       //   Add the right hand entry to the parent.
  right_branch->count--;
  for(int i=0; i<right_branch->count; i++){
                       //   Move right-hand entries to fill the hole.
    right_branch->data[i]=right_branch->data[i+1];
    right_branch->branch[i]=right_branch->branch[i+1];
  }
  right_branch->branch[right_branch->count]=
    right_branch->branch[right_branch->count+1];
}

template<class Record,int order>
void B_tree<Record,order>::move_right(B_node<Record,order>*current,
                                      int position)
/*Pre: current points to a node in a B-tree with more than the minimum number of entries
       in branch position and one too few entries in branch position-1.
  Post: The rightmost entry from branch position has moved into current, which has now
       an entry into the branch position+1.*/
{ B_node<Record,order>*right_branch=current->branch[position+1],
                      *left_branch=current->branch[position];
  right_branch->branch[right_branch->count+1]=
    right_branch->branch[right_branch->count];
  for(int i=right_branch->count ; i> <0; i--){ //   Make room for new entry.
    right_branch->data[i]=right_branch->data[i-1];
    right_branch->branch[i]=right_branch->branch[i-1];
  }
  right_branch->count++;
  right_branch->data[0]=current->data[position];
                                //   Take entry from parent.
```

```
  right_branch->branch[0]=left_branch->branch[left_branch->count--];
  current->data[position]=left_branch->data[left_ branch->count];
}

template< class Record, int order>
void B tree<Record, order>::combine(B_node<Record, order>*current,
                                    int position)
/*Pre: current points to a node in a B tree with entries in the branches position and
       position-1, with too few to move entries.
  Post: The nodes at branches position-1 and position have been combined into one node,
       which also includes the entry formerly in current at index position-1.*/
{ int i;
  B_node<Record, order>*left_branch=current->branch[position-1],
                       *right_branch=current->branch[position];
  left_branch->data[left_branch->count]=current->data[position-1];
  left_branch->branch[++left_branch->count]=right_branch->branch[0];
  for(i=0;i<right_branch->count;i++){
    left_branch->data[left_branch->count]=right branch->data[i];
    left_branch->branch[++left_branch->count]=
                          right_branch->branch[i+1];
  }
  current->count--;
  for(i=position-1;i<current->count;i++){
    current->data[i]=current->data[i+1];
    current->branch[i+1]=current->branch[i+2];
  }
  delete right_branch;
}
```

练习 11.3

E1. 以下列次序将字母表的剩余 6 个字母插入到图 11.10 的最终 B-树中。

$$z,\ v,\ o,\ q,\ w,\ y$$

E2. 将下列元素以给定的次序插入初始为空且阶为(1)3,(2)4,(3)7 的 B-树中：

$$a\quad g\quad f\quad b\quad k\quad d\quad h\quad m\quad j\quad e\quad s\quad i\quad r\quad x\quad c\quad l\quad n\quad t\quad u\quad p$$

E3. 当以合适的次序插入元素时，产生的高度为 3(即 3 层)的 5 阶 B-树的最少元素数是多少?

E4. 画出可以从键 1,2,3,4,5,6,7 和 8 构造出来的所有 5 阶(每个结点有 2～4 个键)B-树。

E5. 如果 B-树中的键不在树叶中，证明它(在自然次序下)的直接前趋和直接后继都在树叶中。

* * * 磁盘访问

E6. 假设磁盘硬件允许我们按需选择磁盘记录的大小，但它从磁盘读取记录的时间是 $a+bd$，其中 a 和 b 是常数，d 是 B-树的阶。(B-树中的一个结点在磁盘上存为一个记录。)设 n 是 B-树中的元素数。为了简化，假设 B-树中的所有结点都是满的(每个结点包含 $d-1$ 个元素)。

(1)解释为什么典型的 B-树操作(如查找、插入)所需要的时间近似于 $(a+bd)\log_d n$。

(2)证明当 d 的值满足 $d(\ln d-1)=a/b$ 时，所需要的时间最少。(注意答案与B-树中的

元素数 n 无关。)(提示:对固定的 a,b 和 n,时间是 d 的函数:$f(d)=(a+bd)\log_d n$。注意:$\log_d n = (\ln\ n)/(\ln\ d)$。为了找出此最小值,计算导数 $f'(d)$ 并将它设为 0。)

(3)假设 a 是 20 毫秒,b 是 0.1 毫秒。(记录很短)找出使时间最小的 d 的(近似)值。

(4)假设 a 是 20 毫秒,b 是 10 毫秒。(记录很长)找出使时间最小的 d 的(近似)值。

*** 遍历

E7. 编写一个遍历链式 B-树的方法,按键的次序(较小的键优先)访问所有的元素。

E8. 递归定义 B-树的前序遍历意味着首先访问根结点中的所有元素,然后以前序从左到右遍历所有的子树。编写一个前序遍历 B-树的方法。

E9. 递归定义 B-树的后序遍历意味着首先以后序从左到右遍历所有根的子树,然后再访问根结点中的所有元素。编写一个后序遍历 B-树的方法。

E10. 删除函数 recursive_search_trec 中的尾部递归并将它并入 search_tree 的非递归版本中。

E11. 重新编写函数 search_node,使用二分查找。

*** B*-树

E12. B*-树是 B-树,在它的每个结点中,除非可能是根,否则至少是 2/3 满的,而不是半满。B*-树的插入在需要时会在兄弟结点之间移动(同删除期间的做法),从而将分裂结点的时间延迟到两个兄弟结点全部已满的时候。这两个结点然后被分裂成 3 个结点,其中的每一个至少是 2/3 满的。

(1)为了保持 B*-树的特性,说明插入算法所需要的修改。

(2)为了保持 B*-树的特性,说明删除算法所需要的修改。

(3)与普通树进行比较,讨论 B*-树相对的优、缺点。

程序设计项目 11.3

P1. 将本节的所有函数合成到用于 B-树的一个菜单驱动演示程序中。如果你足够仔细地设计了 10.2 节项目 P2 的二叉查找树的演示程序,则可以直接替换它的操作包。

P2. 将 B-树的检索和插入函数替换到 10.2 节项目 P5 的信息检索项目中。用输入文本文件和 B-树的阶的各种组合,对 B-树的性能和二叉查找树的性能进行比较。

11.4 红-黑树

11.4.1 介绍

*** B-树结点

在最后一节,我们用顺序表存储 B-树单个结点中的元素。这样做是比较合适的,因为一个结点中的元素数通常相对较少,而且因为我们仿效了可能用于磁盘外部文件的方法,在那儿可能无法利用动态内存,但记录却可以顺序地存储在磁盘上。

*** 二叉树表示

然而,一般来说,可以使用我们希望的任何有序结构去存储每个 B-树结点中的元素,而

小型的二叉查找树证明是一种优秀的选择。我们仅仅需要仔细地区分单个 B-树结点内的链和从一个 B-树结点到另一个 B-树结点的链，因而我们现在以浅灰色曲线画出一个 B-树结点内的链，用直线画出 B-树结点之间的链。图 11.6 显示了以这种方式构造的 4 阶B-树。

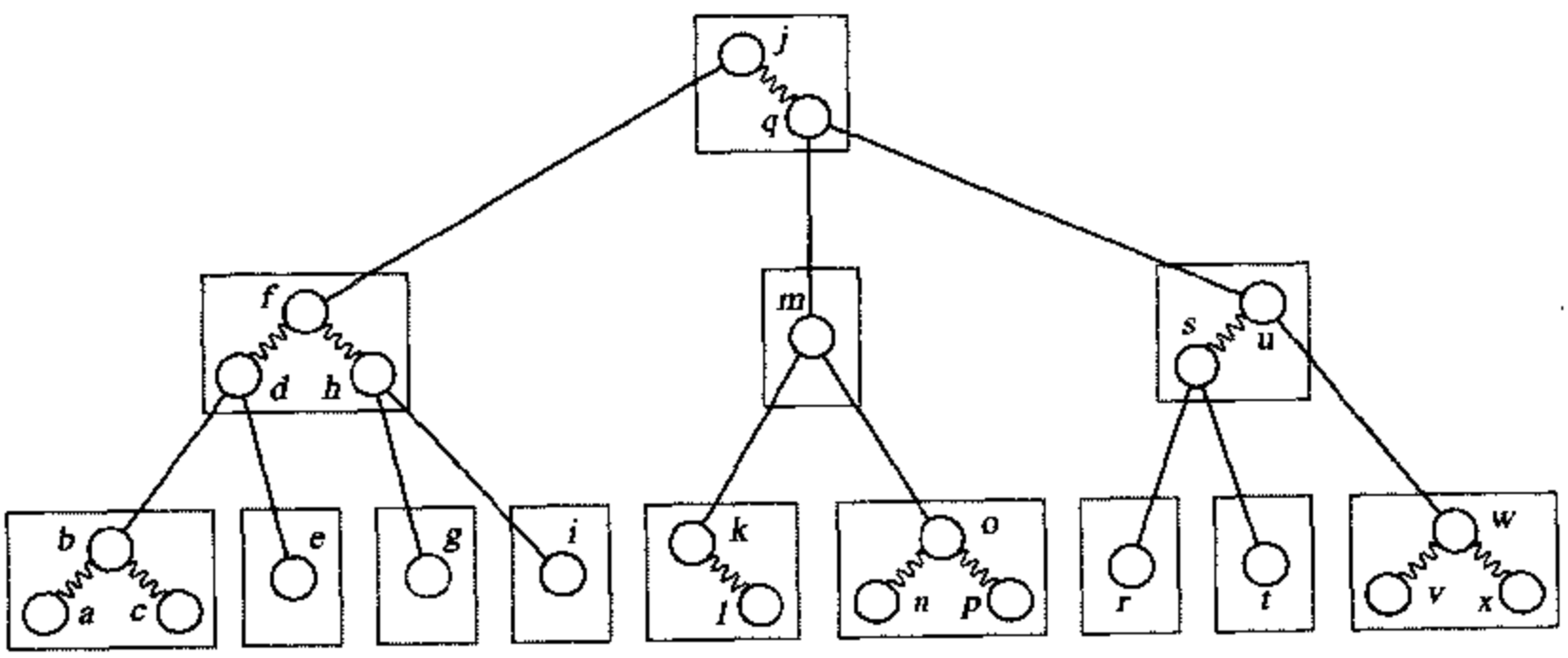
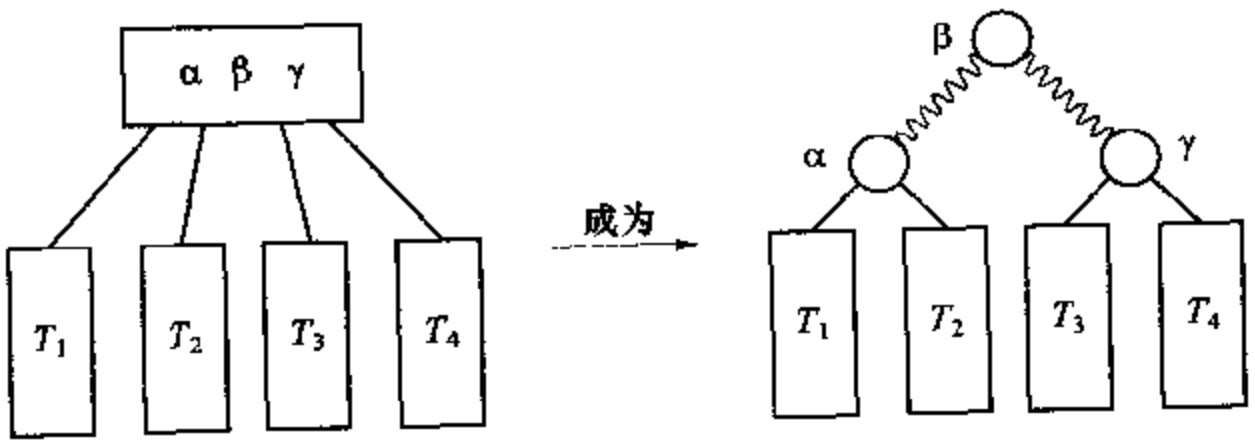


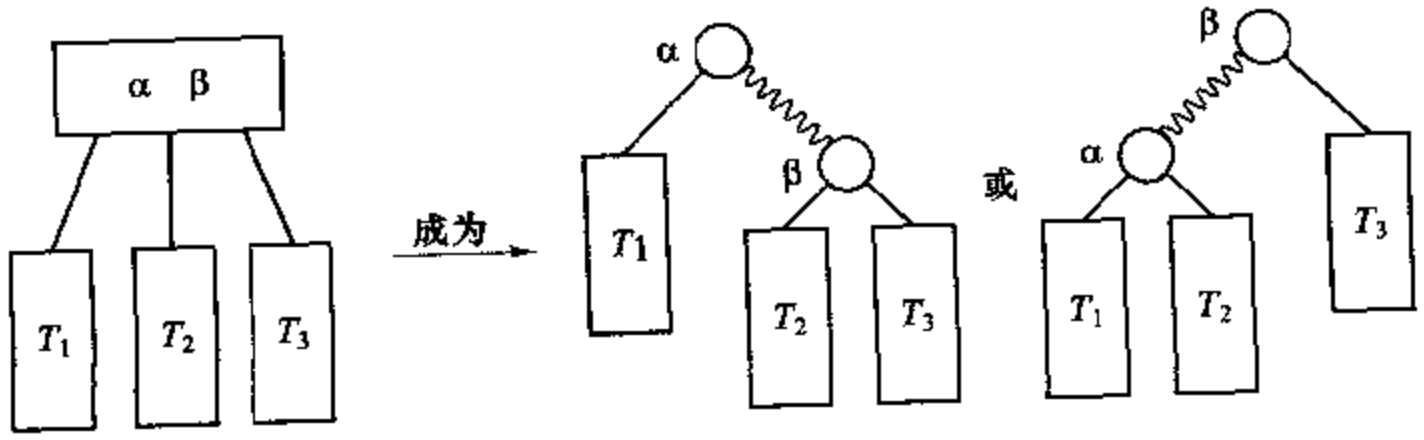
图 11.16　作为二叉查找树的 4 阶 B-树

11.4.2　定义和分析

这种构造方法对于 4 阶 B-树极其有用(如图 11.16)，其中树的每个结点包含 1 个、2 个或 3 个元素。具有 1 个键的结点在 B-树和二叉查找树中是相同的，而具有 3 个元素的结点变换为：



具有两个元素的结点存在两种可能的表示(见下页图)：



如果我们愿意，可以总是只使用这两种表示中的一种，但这样做没有道理，而且我们会发现我们的算法自然地产生这两种形式，因此我们允许用任一种形式表示具有两个元素的B-树结点。

＊＊＊第一个定义

因此我们得到这一节的基本定义：红-黑树是将链涂成红色或黑色的二叉查找树，它以刚才描述的方法从4阶B-树得到。在将B-树转换成红-黑树以后，其使用方式同任何其他的二叉查找树。特别地，如果忽略链的颜色，红-黑树的查找和遍历与普通二叉查找树将完全一样。然而，插入和删除要求我们更仔细地保持基本的B-树结构。因此我们将对B-树的要求转换成对红-黑树的相应要求。

＊＊＊有色结点

然而，首先我们采用另一个符号：将红-黑树的每个结点看成与直接在它之上的链具有同样颜色，因而我们经常说成红结点和黑结点而不说红链和黑链。这样，我们对每个结点仅需要保存一位额外信息去表示它的颜色。

＊＊＊根颜色

由于根的上面没有链，因此根不能按这种方式获得颜色。为了简化某些算法，我们约定：根着成黑色。类似地，我们认为所有的空子树(对应于NULL链)都着成黑色。

＊＊＊黑条件

定义B-树的第一个条件是它的所有空子树在同一层上，这意味着从根到空子树(NULL)的每条简单路径都经过了同样数目的B-树结点。相应的红-黑树对每个B-树结点有一个黑结点(或许有1个或2个红结点)。因此我们得到“黑条件”：

> 从根到空子树的每条简单路径经过同样数目的黑结点。

＊＊＊红条件

B-树满足查找树特性的断言自动地满足了红-黑树，而且对4阶树来说，此定义的剩余部分相当于是说每个结点包含1个、2个或3个元素。我们需要红-黑树上的一个条件，它保证(通过红链)合起来只是3个结点被确定为一个B-树结点，而且3个元素的结点是以我们正使用的平衡形式。这个保证来自于“红条件”：

> 如果结点是红的，则它的双亲存在并且是黑的。

(因为我们要求根为黑，所以红结点的双亲始终存在。)

我们提出一个完全不参照B-树的正式定义，对这个讨论做个总结。

定义 红-黑树是二叉查找树，树中的每个结点或者是红色或者是黑色，而且满足下列条件：

＊＊＊黑条件

1. 每条从根到空子树(NULL链)的简单路径经过同样数目的黑结点。

＊＊＊红条件

2. 如果结点是红的，则它的双亲存在并且是黑的。

从这个定义，可以得出从根到空子树的每条路径的长度都不会超出另一条路径的两倍，因为根据红条件，这样的路径上只有不到一半的结点是红的，而根据黑条件，在每条这样的路径上黑结点的数目是一样的。因此我们得到：

定理 11.2 包含 n 个结点的红-黑树的高度仅是 $2 \lg n$。

* * * 查找性能

因此在每种情况中，查找 n 个结点的红-黑树的时间是 $O(\log n)$。我们将发现插入的时间也是 $O(\log n)$，但首先需要设计相关的算法。

然而，回忆 10.4 节中的 AVL 树，它在最坏情况时的高度大约为 $1.44 \lg n$，平均情况甚至有更小的高度。因此，红-黑树没有达到和 AVL 树一样好的平衡，但这并不表示红-黑树一定比 AVL 树慢，因为 AVL 树可能需要比红-黑树更多的旋转去保持平衡。

11.4.3 红-黑树说明

为了表示红-黑树对象，考虑 C++类说明的几种选择。我们可以回到最初的动机，将红-黑树作为 B-树实现，此时 B-树的结点保存的是查找树而不是顺序表。这个办法要求我们对 B-树的大多数方法和辅助函数进行重新编码，因为原始版本非常依赖于结点元素的顺序表示。因此我们将考察一种可选择的实现，在这里构造的红-黑树类继承了 10.2 节查找树类的特性。

首先我们必须将构成红-黑树的颜色合并到结点中：

```
enum Color {red,black};
template< class Record>
struct RB_node: public Binary_node<Record>{
  Color color;
  RB_node(const Record &new_entry){color=red; data=new_entry;
                                   left=right=NULL; }
  RB_node()                        {color=red; left=right=NULL; }
  void set_ color(Color c)         {color=c; }
  Color get_color() const          {return color; }
};
```

* * * 结点构造函数和方法

为方便起见，我们已包含了红-黑结点的构造函数和其他方法的内联函数。我们看到 **struct** RB_node 与前面 10.4 节中用于保存 AVL 树的结点的 **struct** AVL_node 非常类似：惟一的改变是我们现在要维持颜色信息而不是平衡信息。

* * * 指针访问方法

为了通过指针调用结点方法 get_color 和 set_color，我们需要将对应的虚函数加入基类 **struct** Binary_node。我们加入类似的虚函数去访问 10.4 节里的平衡信息。修改后的结点说明采用如下的形式：

```
template<class Entry>
struct Binary_node {
  Entry data;
```

```
  Binary_node<Entry>*left;
  Binary_node<Entry>*right;
  virtual Color get_color()const { return red;}
  virtual void set_color(Color c){}
  Binary_node()                  { left= right= NULL;}
  Binary_node(const Entry &x)    {data=x;left=right=NULL;}
};
```

就如在10.4节中，一旦做出这些修改，我们就能重用前面用于操纵二叉查找树及其结点的所有方法和函数。特别是，查找和遍历对红-黑树和与二叉查找树来说是一样的。

我们的主要目标是为红-黑树的类创建修正的插入方法，新的方法必须将新的数据插入红-黑树，从而使红-黑树在插入后仍保持其特征。因此我们需要下面的类说明：

```
template<class Record>
class RB_tree:public Search_tree<Record>{
public:
  Error_code insert(const Record &new_entry);
private:                    //  Add prototypes for auxiliary functions here.
};
```

11.4.4 插入

* * *总述

我们以二叉查找树插入的标准递归算法开始，也就是说，我们比较target的新键和根(如果树非空)的键，然后递归地将新元素插入到根的左子树或右子树中。当遇到空子树时这个过程终止，于是我们创建了一个新结点并将它附加到树上去代替空子树。

* * *新结点

这个新结点是红的还是黑的？如果让它为黑，则将增加一条路径(且仅一条路径)上的黑结点数，从面违背了黑条件，因此新结点必须是红的。(同时记住B-树中新元素的插入首先是进入一个已有的结点，这个过程对应于将一个新的红结点附加到红黑树上。)如果新结点的双亲是黑的，则完成插入操作，但如果其双亲是红的，则会导致违背树中的红条件，因为在这条路径上有两个相邻的红结点。插入算法的主要工作是去掉这样的红条件违背，而且我们发现有几种不同的情况需要进行单独处理。

* * *推迟工作

* * *状态变量

然而，如果我们不立即考虑这些情况，而是尽可能地往后推迟此项工作，则我们的算法就非常简单。因此，当我们让一个结点是红的，我们不是立即努力去修整这棵树，而是简单地从递归调用返回，用一个状态指示符指示刚被处理的结点是红的。

* * *双亲结点:红违背

在此返回之后，我们再次处理其双亲结点。如果它是黑的，则红-黑树的条件满足，过程终止。如果它是红的，则我们再次不立即努力去修整这棵树，而是设置状态变量以指示现在总共有两个红结点，然后简单地从递归调用返回。在这种情况下，同时使用状态变量指示这两个红结点是否作为左孩子或者右孩子相关是有用的。

当从第二个递归调用返回后，我们开始处理祖父结点。在这儿，根总是黑结点的约定显示出其用处：因为双亲结点是红的，因此它不可能是根，从而它的祖父结点存在。此外，这个祖父结点保证是黑的，因为它的孩子（双亲结点）是红的，因此对红条件的仅有的违犯是在树下更远的地方。

＊＊＊祖父结点：恢复

最后，在祖父结点的递归层，能够对树进行变换以恢复红-黑条件。我们只检查双亲是祖父的左孩子的情况，而它与右孩子的情况是对称的。需要根据祖父的其他（右）孩子（即初始结点的“阿姨”或“叔叔”）的颜色区分这两种情况。

＊＊＊黑阿姨

首先假设这个阿姨是黑的，这种情况也覆盖了阿姨结点不存在的可能。（我们在前面已认为空子树是黑的。）然后通过向右的单或双旋转恢复红-黑树的特征，如图 11.17 的最初两部分显示。需要验证的是，在这两个图中，旋转（和相关的颜色变化）去掉了违背的红条件，并且由于不改变树的任一向下路径中黑结点的数目而保持黑条件。

＊＊＊红阿姨

现在假设阿姨结点是红的，如图 11.17 的最后两部分显示。这儿的变换更简单：没有发生旋转，但它们的颜色改变了：双亲和阿姨结点变成黑的，而祖父结点变成红的。我们需要再次验证树的任何向下路径上的黑结点的数目仍然一样。然而，由于祖父结点变成红的，很可能仍要违背红条件：因为曾祖父结点也可能是红的，因此这个过程不能终止。我们需要设置状态指示符，指示我们有一个新的红结点，从而可能需要继续进行递归。然而任何红条件的违背会在树中向上移两层，而由于根是黑的，因此这个过程最终将终止。这个过程也可能将根从黑的变为红的，因此，在最外层调用，如果必要，我们需要确信根变回黑的。

11.4.5　插入方法的实现

现在将这个过程用于实现插入并将它转化成 C++形式。照常，我们将在递归函数中完成几乎所有的工作，因此插入方法只做一些设置和错误检查，这个工作最重要的部分是跟踪指示递归插入结果的状态。对于此状态指示符，我们设置了一个新的枚举类型，如下：

```
enum RB_code {okay, red_node, left_red, right_red, duplicate};
/* These outcomes from a call to the recursive insertion function describe the fol-
   lowing results:
   okay: The color of the current root (of the subtree) has not changed; the tree now
         satisfies the conditions for a red-black tree.
   red node: The current root has changed from black to red; modification may or may not
             be needed to restore the red-black properties.
   right_red: The current root and its right child are now both red; a color flip or
              rotation is needed.
   left_red: The current root and its left child are now both red; a color flip or
             rotation is needed.
   left_red: The current root and its left child are now both red; a color flip or
             rotation is needed.
   duplicate: The entry being inserted duplicates another entry; this is an error.
              */
```

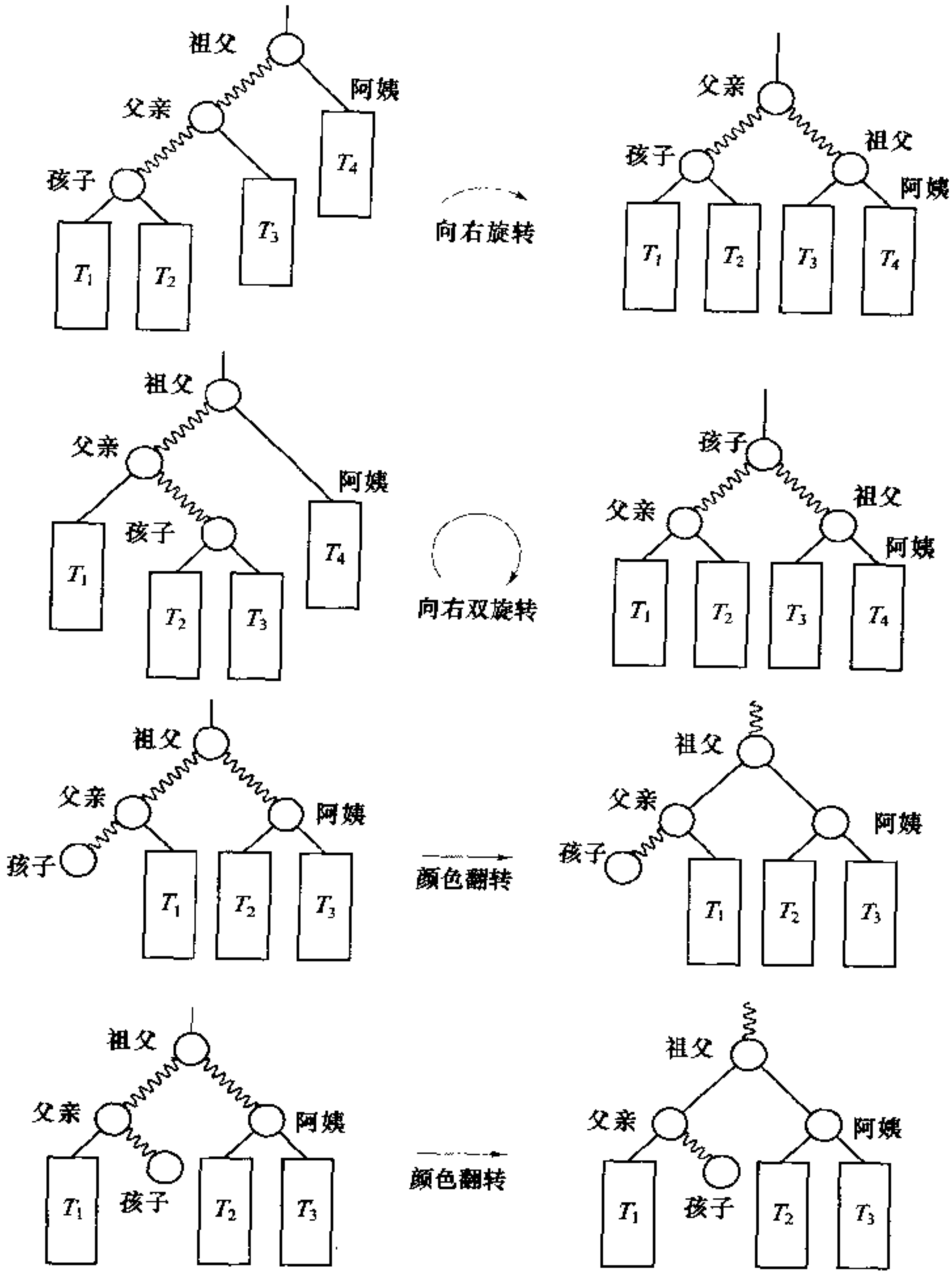


图 11.17 恢复红-黑条件

插入方法的另一项惟一的任务是强迫根着成黑色。因此我们有：

```
template< class Record>
Error_code RB_tree<Record>::insert(const Record &new_entry)
/*Post: If the key of new_entry is already in the RB_tree, a code of duplicate_error is
        returned. Otherwise, a code of success is returned and the Record new_entry is
        inserted into the tree in such a way that the properties of an RB tree have been
        preserved.
  Uses: Methods of struct RB_node and recursive function rb_insert.*/
```

```
{
  RB_code status=rb_insert(root,new_entry);
  switch(status){              //  Convert private RB_code to public Error_code.
    case red_node:             //  Always split the root node to keep it black.
      root->set color(black); /*Doing so prevents two red nodes at the top of the tree
            and a resulting attempt to rotate using a parent node that does not
            exist.* /
    case okay:
       return success;
    case duplicate:
       return duplicate_error;
    case right_red:
    case left_red:
      cout<< "WARNING:Program error detected in RB tree::insert"<<endl;
      return internal_error;
  }
}
```

递归函数 rb_insert 完成实际的插入，它以通常的方式查找树，并继续下去直到遇到空子树，在这里实际的插入被调用 RB_node 的构造函数所代替。当函数从递归调用跳出时，它或者使用 modify_left 或者使用 modify_right 去执行图 11.17 中显示的条件要求的和由 RB_code 状态指定的旋转和颜色翻转。

```
template<class Record>
RB_code RB_tree<Record>::rb_insert(Binary_node<Record>*&current,
                                   const Record &new_entry)
/*Pre: current is either NULL or points to the first node of a subtree of an RB_tree
  Post: If the key of new_entry is already in the subtree, a code of duplicate is
        returned.Otherwise,the Record new_entry is inserted into the subtree pointed
        to by current. The properties of a red-black tree have been restored, except
        possibly at the root current and one of its children,whose status is given by the
        output RB_code.
  Uses: Methods of class RB_node, rb_insert recursively,modify_left,and modify_right.
        */
{
  RB_code status,
          child status;
  if(current==NULL){
     current=new RB_node<Record>(new_entry);
     status=red_node;
  }
   else if(new_entry==current->data)
     return duplicate;
   else if(new_entry<current->data){
     child_status=rb_insert(current->left,new_entry);
     status=modify_left(current,child_status);
    }
    else {
     child_status=rb insert(current->right,new_entry);
     status=modify_right(current,child_status);
    }
    return status;
 }
```

函数 modify_left 更新状态变量并识别在图 11.17 中显示的要求旋转或颜色翻转的情况。正是在这个函数中，我们推迟恢复红-黑特性的决定获得了成功。当 modify_left 被调用时，我们知道插入是在当前结点的左子树中进行，我们也知道它的颜色，而且通过使用 RB_code 状态变量，我们还知道插入所进入的子树的条件。通过使用所有这些信息，我们能正确地决定需要什么操作(如果需要任何操作的话)去恢复红-黑特性。

```
template<class Record>
RB_code RB_tree<Record>::modify_left(Binary_node<Record>*&current,
                                     RB_code &child_status)
/*Pre: An insertion has been made in the left subtree of *current that has returned the
       value of child_status for this subtree.
  Post: Any color change or rotation needed for the tree rooted at current has been made,
       and a status code is returned.
  Uses: Methods of struct RB_node, with rotate_right, double_rotate_right, and flip_
       color.*/
{
  RB_code status=okay;
  Binary_node<Record>*aunt=current->right;
  Color aunt_color=black;
  if(aunt!=NULL)aunt_color=aunt->get_color();
  switch(child_status){
  case okay:
    break;                          //  No action needed,as tree is already OK.
  case red_node:
    if(current->get_color()==red)
      status=left_red;
    else
      status=okay;                  //  current is black,left is red,so OK.
    break;
  case left_red:
    if(aunt_color==black)status=rotate_right(current);
    else                            status=flip_color(current);
  break;
  case right_red:
    if(aunt_color==black)status=double_rotate_right(current);
    else                            status=flip_color(current);
    break;
  }
  return status;
}
```

辅助函数 modify_right 处理图 11.17 中所示情形的镜像情形，它与 modify_left 类似。在图 11.17 中显示了旋转函数和颜色翻转函数的操作，而这些函数都可以留作练习。旋转函数能基于 AVL 树的旋转函数，但对于红-黑树，正确地设置颜色和状态指示符很重要，如图 11.17 所示。

11.4.6 删除结点

就像 B-树中删除结点比插入结点复杂得多的事实一样，从红-黑树中删除比插入更复

杂。因为插入产生了新的红结点，这可能违背红条件，我们需要在插入后专心仔细地注意恢复红条件。另一方面，红结点的删除几乎不引起任何困难，而黑结点的删除可能引起黑条件的违背，而为了恢复树的黑条件，需要考虑许多特殊的情况，这些情况太多，以致我们甚至无法尝试略述所需要的步骤。更多的关于删除算法的信息请参考本章结束的参考书目。

练习 11.4

E1. 将键 c,o,r,n,f,l,a,k,e,s 插入到初始为空的红-黑树。

E2. 将键 a,b,c,d,e,f,g,h,i,j,k 插入到初始为空的红-黑树。

E3. 找出一棵二叉查找树，使得无法对它的结点着色，从而不能使它成为一棵红-黑树。

E4. 找到一棵非 AVL 树的红-黑树。

E5. 证明任意 AVL 树能将它的结点着色而成为红-黑树。证明下面这个更强的命题更简单：高度为 h 的 AVL 树能对它的结点像红-黑树一样着色，在每条到空子树的路径上恰有$\lceil h/2\rceil$个黑结点，并且如果 h 是奇数时，根的两个孩子都是黑的。

程序设计项目 11.4

P1. 通过编写下列缺少的函数完成红-黑树的插入：

(1)modify_right　　(4)rotate_right

(2)flip_color　　(5)double_rotate_left

(3)rotate_left　　(6)double_rotate_right

确信在每个函数的末尾，已正确地设置了受影响的结点颜色，且返回的 RB_code 正确地指示了当前条件。通过包含对非法情形的全面的错误检测，可以简化纠正错误工作的过程。

P2. 将红-黑插入函数代入 10.2 节项目 P2 中用于二叉查找树的菜单驱动演示程序中，从而得到红-黑树的演示程序。可以不用考虑尚未实现的删除。

P3. 将红-黑插入函数代入 10.2 节项目 P5 的信息检索项目中。对于各种组合的输入文本文件，比较红-黑树的和其他查找树的性能。

启示和易犯的错误

1. 对于问题建模和数据组织，树都是灵活而功能强大的结构。在问题求解和算法设计中使用树时，在考虑实现细节前，首先确定所需要的树的种类(有序树、有根树、自由树或二叉树)。
2. 大多数树能通过递归进行描述，因此它们关联的算法通常最好用递归方式设计。
3. 对信息检索问题，在选择将使用的数据结构时，要考虑规模、数目、记录的位置及元素的类型和结构。对于小型记录或较少数目的元素，要使用高速内存，二叉查找树可能证明是足够用的。对于磁盘文件的信息检索，运用多路分支的方法如 trie、B-树和哈希表通常是最优的。trie 特别适合用于键的结构是一个符号序列和键集在所有可能的键集中相对密集

这样的应用。对于其他的应用，把键看作单个单元的方法通常证明是出众的。B-树及其各种推广和扩展，能有效地应用于许多关于外部信息检索的问题中。

复习题

11.1

1. 定义术语(1)自由树；(2)有根树；(3)有序树。
2. 画出具有 3 个顶点的所有不同的：(1)自由树；(2)有根树；(3)有序树。
3. 列举描述果园和二叉树之间对应关系的 3 种方法，并指出其中每种方法的主要目的。

11.2

4. 什么是 trie?
5. 具有 6 层且每个结点中有 5 路分支的 trie 与具有 6 层且除树叶外每个结点有 5 个孩子的有根树有何不同？此 trie 和此树中谁有更少的结点，为什么？
6. 讨论查找 trie 与二叉查找树在速度上的相对优点。

11.3

7. 多路查找树和 trie 有何不同？
8. 什么是 B-树？
9. 当试图将一个新元素插入到 B-树中一个满结点时会发生什么情况？
10. B-树在它的树叶或根上生长吗？为什么？
11. 从 B-树删除一个结点时，什么时候需要合并结点？
12. B-树特别适用于什么目的？

11.4

13. 红-黑树和 B-树之间是什么关系？
14. 陈述黑条件和红条件。
15. 红-黑树的高度和它的大小有什么关系？

进阶参考书目

关于树的十分有用的研究之一是 Knuth 的系列书，从有序树到二叉树的对应出现在第 1 卷的第 332~347 页。第 3 卷的第 471～505 页讨论了多路树、B-树和 trie。

trie 的最早研究见：Edward Fredkin, "Trie memory", *Communications of the ACM* 3 (1960), 490-499.

关于 B-树的原始参考书目是：R. Bayer And E. McCreight, "Organization and maintenance of large ordered indexes", *Acta Informatica* 1(1972), 173-189.

关于B-树的应用及其变种的一个有趣的纵览是：D. Comer, "The ubiquitous B- tree", *Computing Surveys* 11(1979), 121-137.

红-黑树的一种可选择的处理方式，包括删除算法，参见下面这本书，它广泛覆盖了许多不同种类的算法：Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, M. I. T. Press, Cambridge, Mass., and McGraw-Hill, New

York,1990,1028 pages.

下面这本书有红-黑树的另一个删除算法的概述,它给出了更全面的数学分析:Derick Wood, *Data Structures*, *Algorithms*, *and Performance*, Addison-Wesley, Reading, Mass., 1993, pages 353-366.

第 12 章　图

这一章介绍图(graph)这种重要的数学结构，它在社会学、化学、地理和电子工程等各种各样的学科中都有应用。我们将学习用现有的数据结构表示图的方法，并将构造几种处理图的重要的算法。最后，我们看一下使用图本身作为数据结构的可能性。

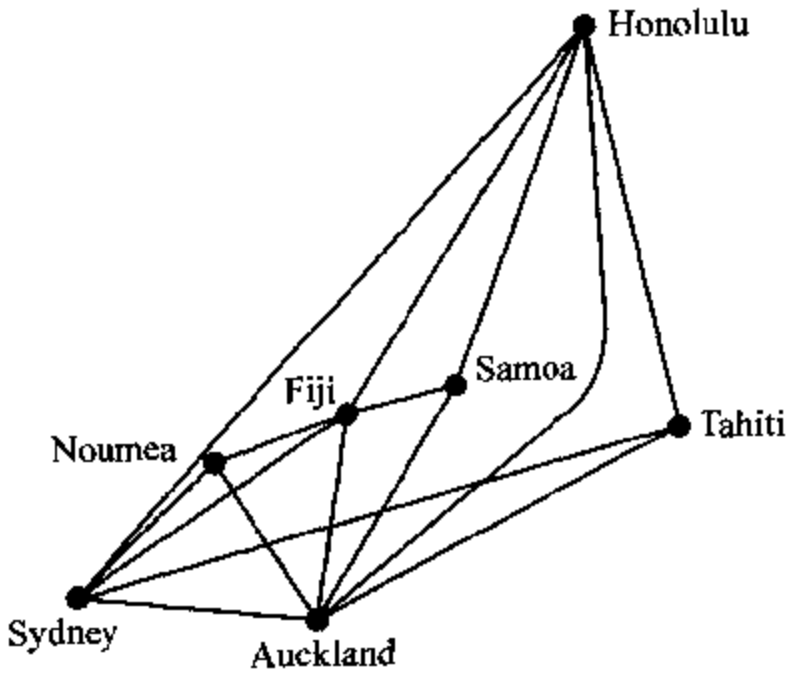
12.1　数学背景

12.1.1　定义和例子

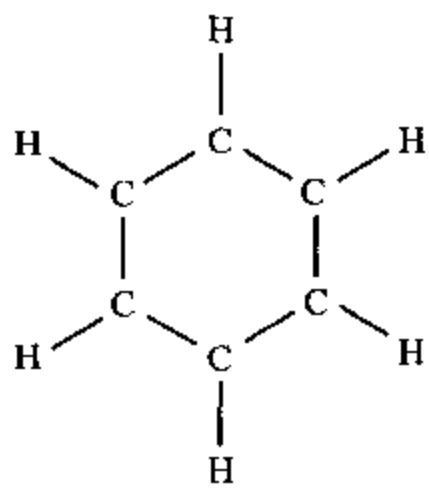
* * * 图和有向图

* * * 画图

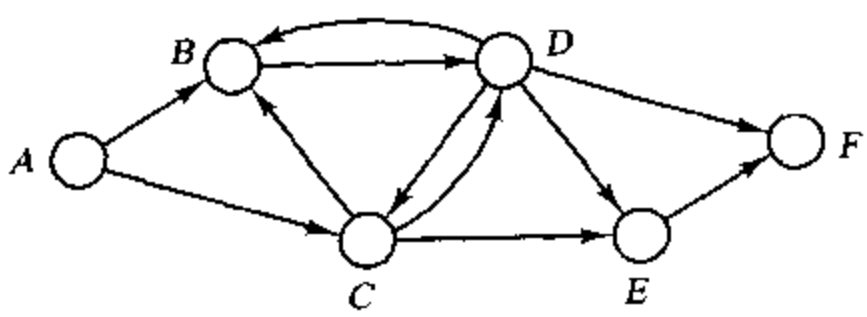
图 G 由集合 V 和 V 中不同顶点对的集合 E 组成，V 的成员称为图 G 的顶点(Vertex)，这些顶点对称为 G 的边(edge)。如果 $e=(v,w)$ 是顶点 v 和 w 组成的边，则说 v 和 w 依赖于 e，而说 e 关联 v 和 w。如果此顶点对是无序的，则称 G 为无向图(undirected graph)；如果此顶点对是有序的，则称 G 为有向图(directed graph)。有向图这个术语在英文表示中经常简化为 digraph，当不带限制条件时，图这个术语通常是指无向图。画出一个图的自然方法是



经过挑选的南太平洋航空路线图



苯分子图



网络中的消息传输图

图 12.1　图的例子

将顶点表示为点或圆，将边表示为连接顶点的线段或弧。如果图是有向的，则线段或弧带有箭头指示方向。图 12.1 显示了图的几个例子。

图 12.1 第一部分中的顶点是地点，连接它们的航空路线是边。在第二部分中，氢和碳原子(用 H 和 C 表示)是顶点，化学键是边。图 12.1 的第三部分显示了有向图，网络结点($A,B,\ldots,F$)是顶点，从一个结点到另一个结点的边具有箭头显示的方向。

*　*　*　应用

图可以作为多种过程或结构的模型，从而体现出它的重要性。城市和连接城市的高速公路形成一个图，电路板上的部件和它们之间的连接也形成一个图。有机化学成分可以看作是一个图，它以原子作为顶点，以原子之间的键作为边。生活在城市中的人可以看作是图的顶点，人之间的熟识关系可以看作是边。在公司里工作的人形成一个有向图，“监督”关系是图的边。同样也能将人看作是一个无向图，图中不同的边描述了“一起劳动”关系。

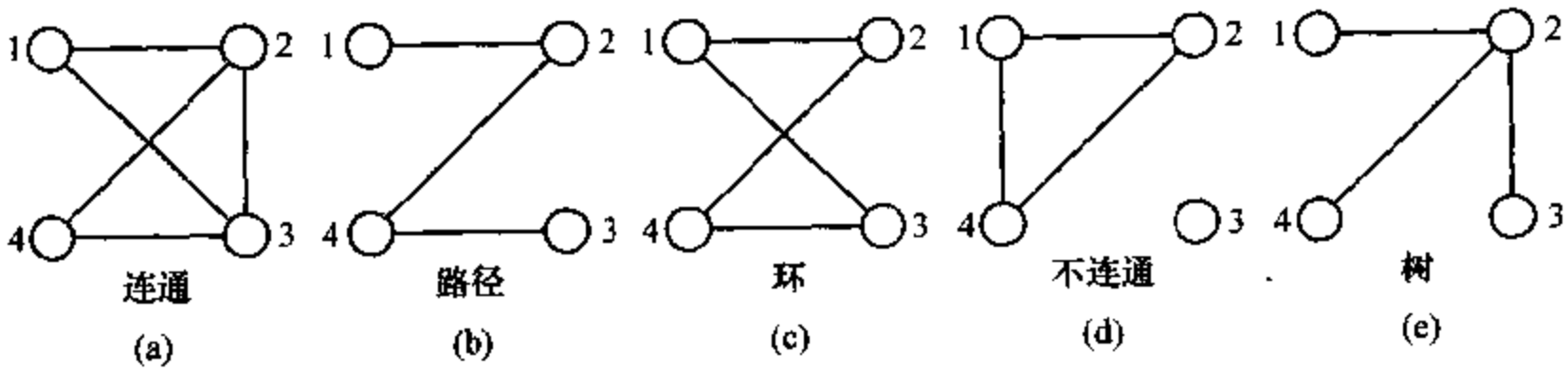


图 12.2　各种各样的无向图

12.1.2　无向图

*　*　*　路径、环和连通

*　*　*　自由树

图 12.2 中显示了几种无向图。对无向图中的两个顶点，如果从一个顶点到另一个顶点存在一条边，则称这两个顶点是邻接的。因此，在(a)部分的无向图中，顶点 1 和 2、3 和 4 是邻接的，但 1 和 4 是非邻接的。路径是由不同顶点构造的一个序列，序列中的每个顶点都与下一个顶点邻接；(b)部分显示了一条路径。环是一条至少包含 3 个顶点的路径，路径上最后一个顶点与第一个顶点邻接；(c)部分显示了一个环。在图中，如果从任一个顶点到任一个其他顶点之间存在一条路径，则称这个图是连通的(connected)。(a)、(b)和(c)部分是连通图，而(d)部分是非连通图。如果图不连通，则连通顶点的最大子集称为连通分量。比如，(c)部分中的非连通图有两个连通分量：第一个由顶点 1，2 和 4 组成，第二个只有顶点 3。图 12.2(e)部分显示了没有环的连通图，实际上，这个图是一棵树，我们采用这种特征做出如下定义：自由树(free tree)定义为没有环的连通无向图。

12.1.3　有向图

*　*　*　有向图和环

对于有向图，可以做类似的定义。在有向图中，要求所有的边或环具有同样的方向，因

此沿着一条边或环意味着总是沿着箭头指示的方向移动。这样的路径(环)称为有向路径(环)。如果从任一个顶点到另一个其他顶点之间存在一条有向路径,则称有向图是强连通的。如果不考虑边的方向且结果得到的无向图是连通的,则称有向图是弱连通的。图 12.3 显示了有向环、强连通有向图和弱连通有向图。

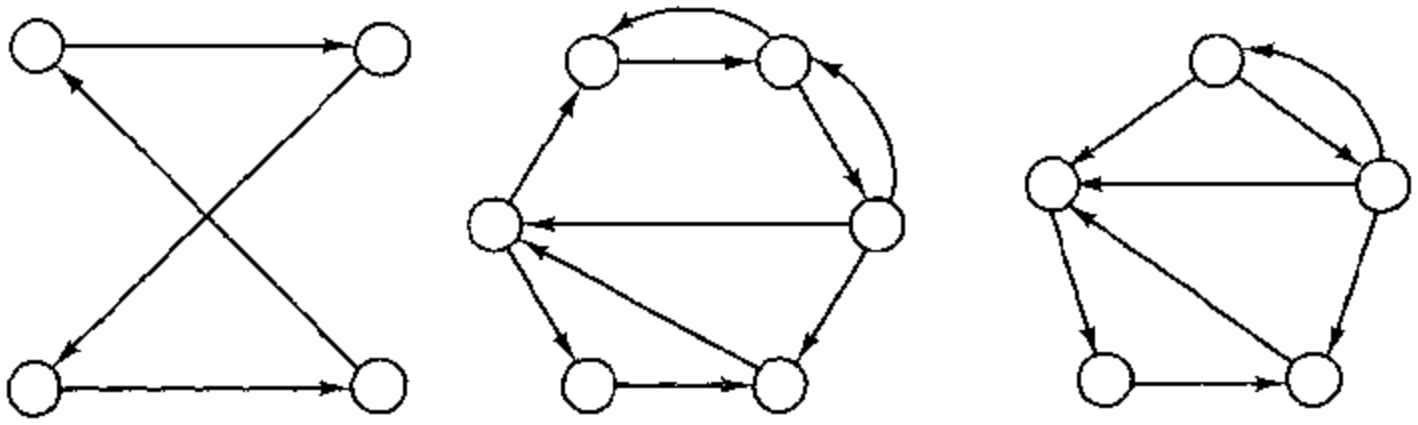


图 12.3 有向图的例子

* * * 多边

* * * 自回路

图 12.3 中(b)和(c)部分的有向图显示了在两个方向上具有有向边的顶点对。因为有向边是有序对,而如果 $v \neq w$,则有序对(v,w)和(w,v)是不同的,在有向图中,允许这样的两条边。在无向图中,由于对应的无序对是相同的,因此至多只有一条边连接一对顶点。类似地,因为要求边上的顶点是不同的,因此不能有一条边从一个顶点回到它本身。然而,有时(虽然不应如此)可以放松这些要求,允许连接一对顶点的多条边和连到顶点本身的自回路。

12.2 计算机表示

如果要编写程序去解决关于图的问题,则首先必须找到将图的数学结构表示为某种数据结构的方法。在通常的使用中,存在几种方法,这些方法因选择用于表示图的抽象数据类型而在根本上是不同的,而且由于抽象数据类型的实现而存在几种变化。换句话说,我们从一个数学系统(图)开始,然后研究如何用抽象数据类型(能够使用集合、表格和表)描述它,最后为我们所选择的抽象数据类型选择实现。

12.2.1 集合的表示

图按照集合来定义,而一开始我们就注意到用集合去确定图的数据表示是很自然的想法。首先,有一个顶点集;其次,边是顶点对的集合。我们不是尝试直接表示这个顶点对的集合,而是分别考虑附加到每个顶点上的边集而将它分成几部分。换句话说,对图中所有的顶点 v,我们通过保存包含 v 的边集 E_v,或同样地通过保存与 v 邻接的所有顶点集 A_v,记录图中所有的边。实际上,我们能利用这个思想提出一个关于图的新的、等价的定义:

定义　有向图 G 由集合 V 和对所有 $v \in V$，V 的子集 A_v 组成，其中 V 称为 G 的顶点集，A_v 称为与 v 邻接的顶点集。

我们能用下面的规则从子集 A_v 利用边重构有序对：当且仅当 $w \in A_v$，(v,w) 对是一条边。然而，在顶点集上操作比在边对上操作会更简单。此外，这个新的定义对有向图和无向图都有用。图是无向的意味着它满足下面的对称特性：$w \in A_v$ 蕴含着对所有的 $v, w \in V$，$v \in A_w$。这个特性可以更加非正式地重新描述为：它意味着 v 和 w 之间的一条无向边能被看作由两条有向边组成，其中一条是从 v 到 w，另一条是从 w 到 v。

1. 集合的实现

* * * 集合作为布尔数组

我们有两种通用方法在数据结构和算法中实现顶点集。其中一种方法是将集合表示成其元素的表，我们不久要研究这种方法。另一种实现经常称为位串，它对集合中每个潜在的元素保留一个布尔值以指示它是否在集合中。为简单起见，我们认为集合中潜在元素的下标为 0 到 max_set - 1 之间的整数，这里 max_set 表示所允许的最大元素数。无论是采用标准的模板库类 std::bitset<max_set>，还是采用我们自己使用模板参数给出集合中最大潜在成员数的类模板，后一种策略都易于实现。

```
template<int max_set>
struct Set{
  bool is_element[max_set];
};
```

* * * 第一种实现：集合

我们现在能充分地说明图的第一种表示：

```
template<int max_size>
class Digraph{
  int count;                    //  number of vertices, at most max_size
  Set<max_size>neighbors[max_size];
};
```

在此实现中，顶点用整数 0～count - 1 标识。如果 v 是这样一个整数，则数组元素 neighbors[v]是与顶点 v 邻接的所有顶点的集合。

2. 邻接表格

* * * 作为数组的集合

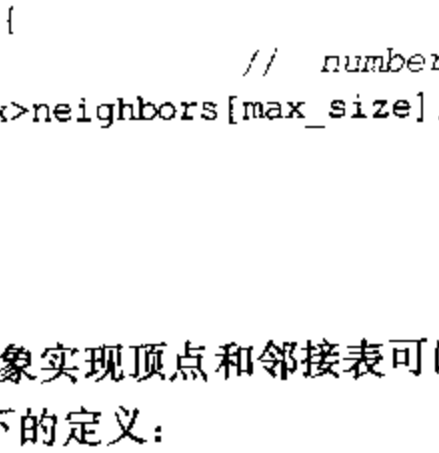
在前面的实现中，结构 Set 实质上是作为布尔型元素的数组实现，每个元素指示相应的顶点是否是集合的成员。如果用 neighbor 集合替换这个数组，我们可以发现，Graph 类定义中的数组 neighbors 能变成一个数组的数组，即，一个二维数组，如下：

* * * 第二种实现：邻接表格

```
template<int max_size>
class Digraph{
  int count;                    //  number of vertices, at most max_size
  bool adjacency[max_size][max_size];
};
```

* * *意义

表格 adjacency 有一个自然的解释：当且仅当顶点 v 与顶点 w 邻接，adjacency[v][w]为真。如果图是有向的，则我们将 adjacency[v][w]解释成指示从 v 到 w 的边是否在图中，如果图是无向的，则邻接表格一定对称，即对所有的 v 和 w，adjacency[v][w]=adjacency[w][v]。图 12.4 说明了图的邻接集合和邻接表格表示。



邻接集合

顶点	集合
0	{1,2}
1	{2,3}
2	ϕ
3	{0,1,2}

邻接表格

	0	1	2	3
0	F	T	T	F
1	F	F	T	T
2	F	F	F	F
3	T	T	T	F

图 12.4　邻接集合和邻接表格

12.2.2　邻接表

另一种表示集合的方法是作为它的元素的表。为了表示一个图，我们不但有顶点表，而且对每个顶点，还有邻接顶点表。可以考虑使用顺序表或者简单链表的图的实现方法。然而，对更高级的应用，用更复杂的表实现如二叉树或多路树或堆经常是有用的。注意，在前面的表示中采用下标区分顶点，我们事实上已经将顶点集作为顺序表实现，但现在我们要对顺序表或链表的使用进行深思熟虑的选择。

1. 基于表的实现

通过用表去替换前面的 neighbors 集合，我们得到基于表的实现，此实现能使用顺序表或者用链表。图 12.5(b)部分说明了顺序版本，图 12.5(c)部分说明了链式版本。

* * *第三种实现：表

```
typedef int Vertex;
template<int max_size>
class Digraph{
  int count;                    //  number of vertices, at most max_size
  List<Vertex>neighbors[max_size];
}
```

2. 链式实现

用链式对象实现顶点和邻接表可以得到最大的灵活性。图 12.5(a)部分说明了这种实现，并引出如下的定义：

* * *第四种实现：链式顶点和边

```
class Edge;                     //  forward declaration
class Vertex{
  Edge*first_edge;              //  start of the adjacency list
```

```
  Vertex*next_vertex;     //  next vertex on the linked list
};
  class Edge{
  Vertex*end_vertex;      //  wertex to which the edge points
  Edge*next_edge;         //  next edge on the adjacency list
}
  class Digraph{
  Vertex*first_vertex;    //  header for the list of vertices
};
```

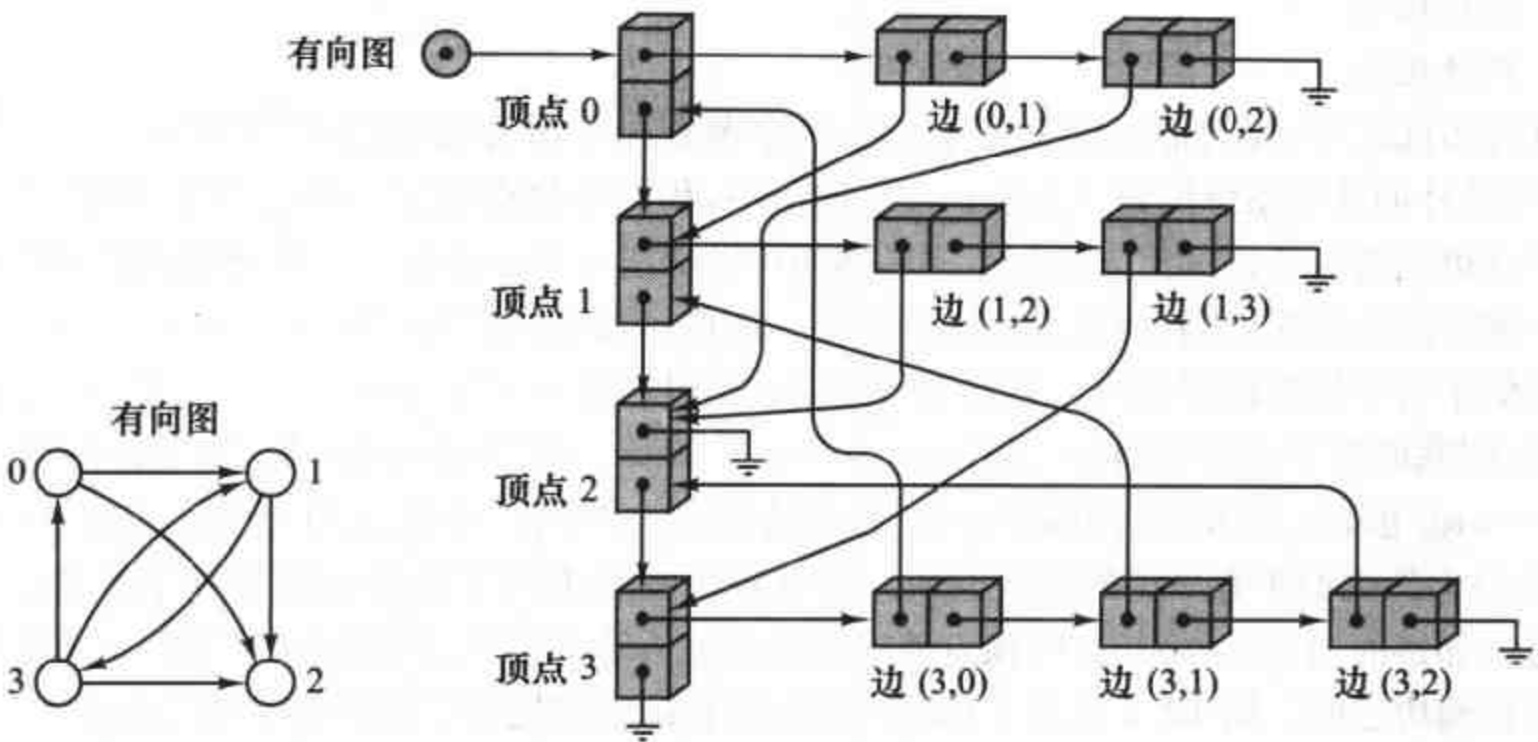


图 12.5　图的表实现

12.2.3　信息域

＊＊＊网络，权

图的许多应用不仅需要在各种表示中说明的邻接信息，而且需要具体到每个顶点或每条边的更多的信息。在链式表示中，此信息能作为合适记录内的附加成员包含进来，而在顺序表示中，信息是通过将数组元素变成记录包含进来。一个特别重要的例子是网络，它定义

为一个图，在这个图中，每条边都附有一个数字权。对网络的许多算法来说，最好的表示方法是邻接表格，其中元素是权而不是布尔值。我们将在这一章的后面再讨论这个主题。

12.3 图的遍历

12.3.1 方法

＊＊＊深度优先

＊＊＊广度优先

在许多问题中，我们希望以某种系统的次序考查图中所有的顶点，就像在学习二叉树时，我们设计的几种系统的遍历方法。在树遍历中，我们通常从根顶点开始；而在图中，没有任何结点可以特殊地挑选出来，因此，图的遍历可以从任一顶点开始。虽然在访问图的顶点时有许多可能的次序，但有两种方法极其重要。图的深度优先遍历(Depth - first traversal)大致类似于有序树的前序遍历。假设遍历刚刚访问过顶点 v，设 $w_1, w_2, \ldots, w_k$ 是与 v 邻接的顶点，则我们接下来将访问 w_1 并使 $w_2, \cdots\cdots, w_k$ 等待。在访问 w_1 后，我们在返回去遍历 $w_2, \cdots\cdots, w_k$ 前，首先遍历所有的与 w_1 邻接的顶点。图的广度优先遍历(Breadth - first traversal)大致类似于有序树的层次遍历。如果遍历刚刚访问过顶点 v，则它接下来会访问所有与 v 邻接的顶点，并将与这些顶点邻接的顶点放入等待表中，在访问完所有与 v 邻接的顶点后再遍历它们。图 12.6 显示了深度优先遍历和广度优先遍历访问图中顶点的次序。

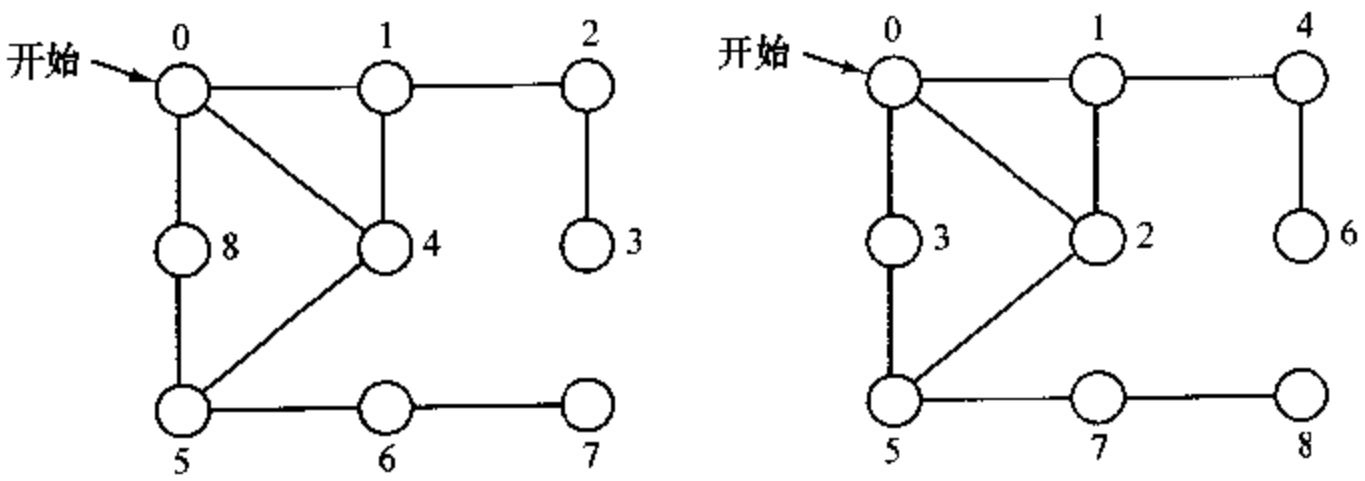


图 12.6 图的遍历

12.3.2 深度优先算法

深度优先遍历自然地描述为一个递归算法。当到达顶点 v 时，它的操作是：

访问(v)；
for(与 v 邻接的每个顶点 w)
遍历(w)；

＊＊＊复杂化

然而，在图的遍历中，出现了树的遍历中不可能出现的两个困难。第一，图可能包含环，因此遍历算法可能又一次到达同样的顶点。为了防止无限循环，我们因此引入一个布尔型

的数组 visited。在访问 v 前，我们将 visited[v] 设置为 true，并在处理 w 前检查 visited[w] 的值。第二，图可能不连通，因此遍历算法可能不能从单个的起始点到达图中所有的顶点。因此，我们将操作放入一个循环中，这个循环在所有的顶点上运行以确保访问了图的所有成分。由这些细化，我们得到下面的关于深度优先遍历的算法概要。更多的细节依赖于图和顶点的实现的选择，因此我们将它们推后到应用程序再行介绍。

* * * 主函数的概要

```
template<int max_size>
void Digraph<max_size>::depth_first(void(*visit)(Vertex &)) const
/*Post: The function *visit has been performed at each vertex of the Digraph in depth-
        first order.
  Uses: Method traverse to produce the recursive depth first order.*/
{ bool visited[max_size];
  Vertex v;
  for(all v in G) visited[v]=false;
  for(all v in G) if(! visited[v])
    traverse(v,visited,visit);
}
```

递归在辅助函数 traverse 中执行。由于 traverse 需要访问图的内部结构，因此它将是 Digraph 类的成员函数。而且，由于 traverse 仅是一个辅助函数，用于方法 depth_first 的构造中，因此它对 Digraph 类应是私有的。

* * * 递归遍历的算法概要

```
template<int max_size>
void Digraph<max_size>::traverse(Vertex &v,bool visited[],
                                 void(*visit)(Vertex &)) const
/*Pre: v is a vertex of the Digraph.
  Post: The depth-first traversal, using function *visit, has been completed for v and
        for all vertices that can be reached from v.
  Uses: traverse recursively.*/
{ Vertex w;
  visited[v]=true;
  (*visit)(v);
  for(all w adjacent to v)
    if (! visited[w])
      traverse(w,visited,visit);
}
```

12.3.3　广度优先算法

* * * 栈和队列

因为使用递归和用栈进行程序设计实质上是一样的，因此可以用 Stack 描述深度优先遍历，我们将所有与正访问的顶点相邻接的且未访问的顶点放入 Stack 中，而从 Stack 中出栈去找到要访问的下一个顶点。除了需要使用 Queue 而不是 Stack 外，广度优先遍历算法与结果得到的深度优先遍历算法十分类似。它的算法概要如下：

* * * 广度优先遍历的算法概要

```
template< int max_size>
void Digraph<max_size>::breadth_first(void(*visit) (Vertex &)) const
/* Post: The function*visit has been performed at each vertex of the Digraph in breadth-
        first order.
  Uses: Methods of class Queue.*/
{ Queue q;
  bool visited[max_size];
  Vertex v, w, x;
  for(all v in G) visited[v]=false;
  for(all v in G)
    if(! visited[v]){
      q.append(v);
      while(! q.empty()){
        q.retrieve(w);
        if (! visited[w]){
           visited[w]=true;
           (*visit) (w);
           for(all x adjacent to w)
            q.append(x);
        }
        q.serve();
      }
    }
  }
```

12.4 拓扑排序

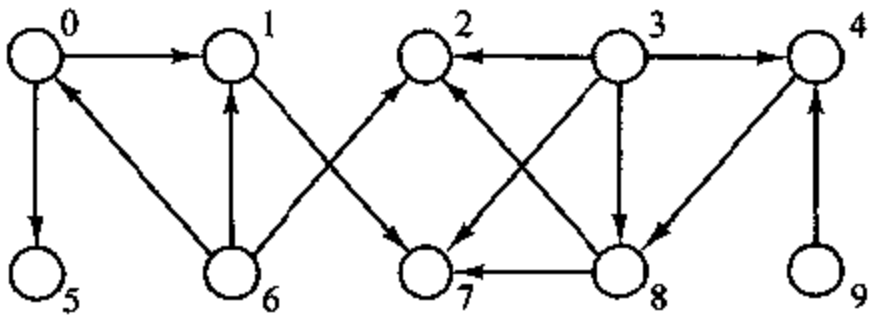
12.4.1 问题

* * * 拓扑次序

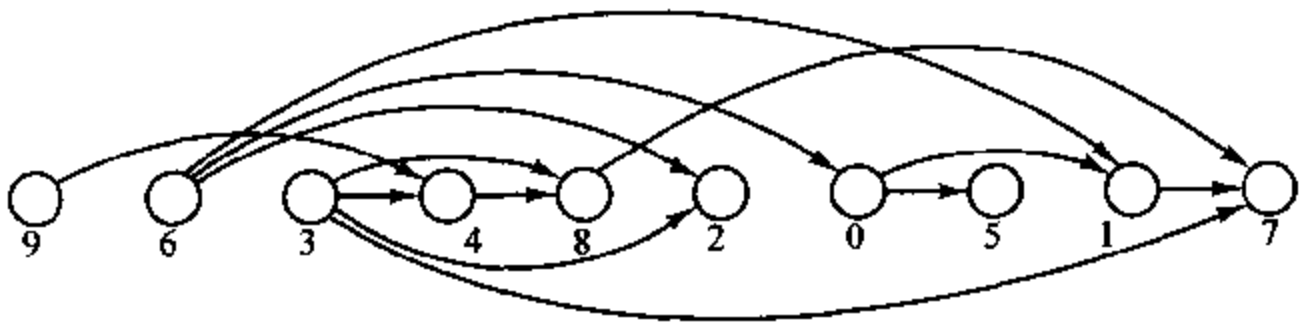
如果 G 是一个无有向环的有向图，则 G 的拓扑次序是 G 中所有顶点的一个顺序排列，使得对所有的顶点 $v, w \in G$，如果有一条从 v 到 w 的边，则 v 在顺序排列中先于 w。在整个这一节，我们只考虑没有有向环的有向图。术语 acyclic 经常用于表示无环图。

* * * 应用

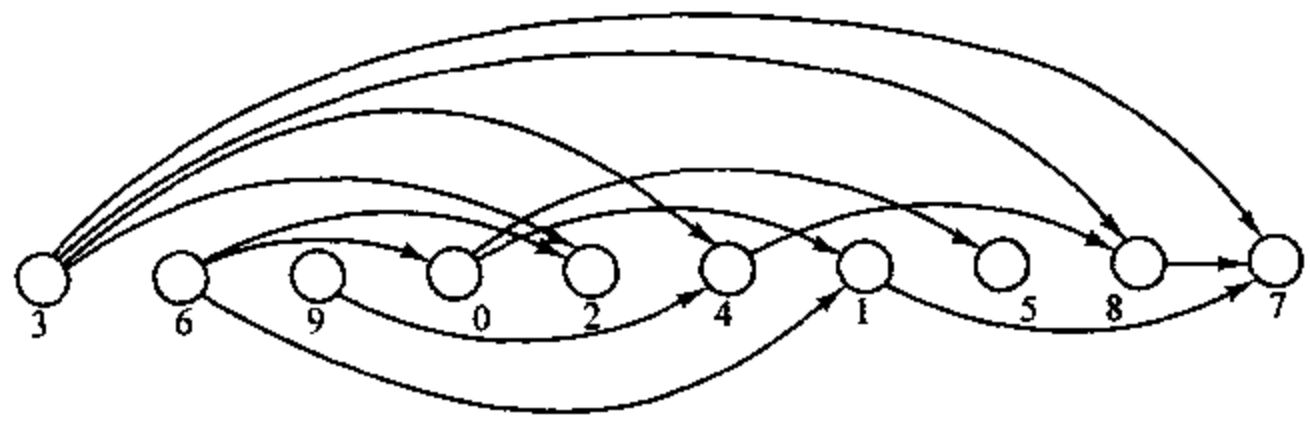
这样的图会在许多问题中出现。作为拓扑次序的第一个应用，我们将大学里现有的课程看作是有向图的顶点，如果第一门课程是第二门课程的先修课程，则从第一门课程到第二门课程有一条边。拓扑次序是所有课程的一个排列，使得一门课程的所有先修课程都在它之前出现。另一个例子是技术术语的汇编，我们要对它进行排序，使得每个术语都不会在本身被定义之前用于其他定义中。类似地，课本的作者对书中的主题使用拓扑次序去安排。在图 12.7 中显示了有向图的两种不同的拓扑次序。作为图遍历算法的一个例子，我们将设计出生成无环有向图顶点的拓扑排序的函数。我们将设计两种方法：第一种方法使用深度优先遍历，第二种方法使用广度优先遍历，并将两种方法都运用到基于表实现的 Digraph 类的对象。因此，我们将采取下面的类说明：



无环有向图



深度优先次序



广度优先次序

图 12.7　有向图的拓扑次序

* * * 图表示

```
typedef int Vertex;

template<int graph_size>
class Digraph {
public:
  Digraph();
  void read();
  void write();
//   methods to do a topological sort
  void depth_sort(List<Vertex>&topological order);
  void breadth_sort(List<Vertex>&topological_order);
private:
  int count;
  List<Vertex>neighbors[graph_size];
  void recursive_depth_sort(Vertex v,bool visited[],
                            List<Vertex>&topological_order);
};
```

辅助成员函数 recursive_depth_sort 将由方法 depth_sort 使用。两种排序方法都会创建一个给出顶点拓扑次序的表。

12.4.2 深度优先算法

* * * 策略

在拓扑次序中，有向图的每个顶点必须出现在其后继所有顶点之前。因此对深度优先拓扑次序，我们从查找没有后继的顶点开始并将它放到序的最后。通过递归，在将一个顶点的所有后继放到拓扑次序中以后，能够将这个顶点自身放到它的所有后继之前的位置中。由于我们首先排列最后的顶点，因此能够重复地将顶点加到 List topologic_order 的开始处。这个方法是上一节中设计的通用深度优先遍历过程的直接实现。

```
template< int graph_size>
void Digraph<graph_size>::depth_sort(List<Vertex>&topological_order)
/*Post: The vertices of the Digraph are placed into List topological_order with a depth
        first traversal of those vertices that do not belong to a cycle.
  Uses:Methods of class List, and function recursive_depth_sort to perform depth first
        traversal.*/
{
  bool visited[graph_size];
  Vertex v;
  for(v=0; v<count; v++)visited[v]=false;
  topological_order.clear();
  for(v=0; v<count; v++)
    if(!visited[v])           //   Add v and its successors into topological order.
      recursive_depth_sort(v,visited,topological_order);
}
```

执行递归的辅助函数 recursive_depth_sort 基于通用函数 traverse 的算法概要，它首先将 v 的所有后继放到拓扑次序中各自的位置上，然后将 v 放到序列中。

```
template<int graph_size>
void Digraph<graph_size>::recursive_depth_sort(Vertex v, bool*visited,
                          List<Vertex>&topological_order)
/*Pre: Vertex v of the Digraph does not belong to the partially completed List to _
       pological_order.
  Post:All the successors of v and finally v itself are added to topological _order with
       a depth-first search.
  Uses:Methods of class List and the function recursive_depth_sort.*/
{
  visited[v]=true;
  int degree=neighbors[v].size();
  for(int i=0; i<degree; i++){
    Vertex w;                 //   A(neighboring) successor of
    neighbors[v].retrieve(i,w);
    if(!visited[w])           //   Order the successors of w.
      recursive_depth_sort(w,visited,topological_order);
  }
  topological_order.insert(0,v); //   Put v into topological_order.
}
```

* * * 性能

由于这个算法仅对图中每个结点访问一次，且沿着每条边一次，它不做查找，所以它的运行时间是 $O(n+e)$，这里 n 是图中的顶点数，e 是图中的边数。

12.4.3　广度优先算法

* * * 方法

在无环有向图的广度优先拓扑次序中，我们从查找拓扑次序中的第一个顶点开始，然后运用每个顶点在拓扑次序中必须在它的后继之前出现这一事实。首先出现的顶点是那些非任何其他顶点的后继的顶点。为了找出这些顶点，我们建立一个数组 predecessor_count，它在下标 v 处的元素是顶点 v 的直接前趋数。非后继的顶点是那些没有前趋的顶点。因此我们在初始化广度优先遍历时，将这些顶点放入待访问的顶点队列 Queue 中。在每个顶点被访问时，就将它从 Queue 中删除，为它分配拓扑次序中下一个可用的位置（从序的开头开始），然后将它的每个直接后继的前趋计数减 1 以免进一步考虑。当这些计数中的某一个达到 0 时，其相应顶点的所有前趋都已经得到访问，于是可以处理顶点本身，因此将它加入到 Queue 中。从而我们得到下面的函数：

```
template<int graph_size>
void Digraph<graph_size>::breadth_sort(List<Vertex>&topological_order)
/*Post: The vertices of the Digraph are arranged into the List topological_order which
        is found with a breadth-first traversal of those vertices that do not belong to a
        cycle.
  Uses: Methods of classes Queue and List.*/
{
  topological_order.clear();
  Vertex v,w;
  int predecessor_count[graph_size];
  for(v=0; v<count; v++)predecessor_count[v]=0;
  for(v=0; v<count; v++)
    for(int i=0; i<neighbors[v].size(); i++){
                              //   Loop over all edges v -w.
      neighbors[v].retrieve(i,w);
      predecessor_count[w]++;
    }
  Queue ready_to_process;
  for(v=0; v<count; v++)
    if(predecessor_count[v]==0)
      ready_to_process.append(v);
  while(!ready_to_process.empty()){
    ready_to_process.retrieve(v);
    topological_order.insert(topological_order.size(),v);
    for(int j=0; j<neighbors[v].size(); j++){   //   Traverse successors of v.
      neighbors[v].retrieve(j,w);
      predecessor_count[w]--;
      if(predecessor_count[w]==0)
         ready_to_process.append(w);
    }
    ready_to_process.serve();
```

```
    }
}
```

* * * 性能

这个算法要求其中一个用于处理队列的软件包。此队列能以第 3 章和第 4 章中描述的任何一种方法实现。因为 Queue 中的元素将是顶点，所以我们应在 breadth_sort 的实现前加上一个说明：**typedef** Vertex Queue_entry;。如同深度优先遍历一样，广度优先函数所需要的时间是 $O(n+e)$，这里 n 是有向图中的顶点数，e 是边数。

12.5 贪心算法：最短路径

12.5.1 问题

* * * 最短路径

图的另一个应用需要稍微复杂的推理，我们考虑下面这个问题。给定有向图 G，它的每条边附有一个非负的权；换句话说，G 是一个有向网络。我们的问题是找到从一个顶点 v 到另一个顶点 w 的路径，使得路径上权的和尽可能小。我们称这样的路径为最短路径，即使这里的权可能是指代价、时间或者其他不是距离的量。比如，可以把图看作是航空路线图，每个顶点表示一个城市，每条边上的权表示从一个城市飞到另一个城市的成本。然后我们的问题是找到一条从城市 v 到城市 w 的路线，使得总成本最小。考虑图 12.8 中显示的有向图。从顶点 0 到顶点 1 的最短路径经过了顶点 2，它的总成本是 4，这与直接从 0 到 1 的边的成本 5 以及经过顶点 4 的成本 8 相比是最短的。

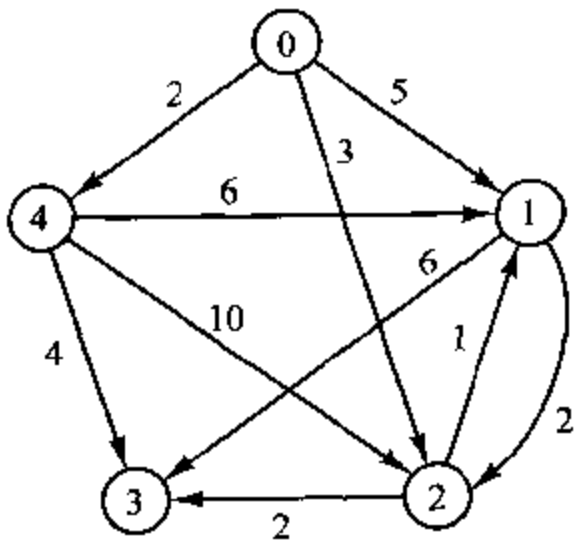


图 12.8 带权有向图

* * * 源点

它表明解决这个问题就像解决从一个称为源点的顶点开始，找出到每个其他顶点而不只是到一个目的顶点的最短路径这个更普遍的问题一样简单。在实现中，源点将作为参数传递。则问题就转化为找出从源点到图中每个顶点的最短路径。我们要求所有的权都是非负数。

12.5.2　方法

算法运行时将维持从源点出发且最短距离已知的顶点集合 S。最初 S 中仅有一个源点，接下来每一步，都会从剩下的顶点中找到一个顶点，使得从源点出发到该顶点的路径最短，然后将它加到 S 中。这个问题要决定的是每一步将哪个顶点加到 S 中。我们认为 S 中的顶点都已标上某种颜色，且认为组成从源点出发到这些顶点的最短路径上的边也已着色。

＊＊＊距离表格

我们将维持一个表格 distance，对每个顶点 v，它给出了从源点沿着已着色的每条边（可能除了最后一条边）到达 v 得到的距离。也即，如果 v 在 S 中，则 distance[v]给出了到 v 的最短路径，并且沿着相应路径的边都已着色。如果 v 不在 S 中，则 distance[v]给出了从源点到 S 中某个顶点 w 的路径长加上从 w 到 v 的边的权值，且这条路径上除了最后一条边以外的所有的边都已着色。在初始化表格 distance 时，如果从源点到 v 的边的权存在，则 distance[v]初始为该权值，否则为无穷大。

＊＊＊贪心算法

＊＊＊验证

为了确定下一步要加到 S 中的顶点，如果 v 不在 S 中，则我们选择记录在表格 distance 中具有最小距离的顶点 v 时运用了贪心准则。对这个顶点 v，必须证明记录在 distance 中的距离确实是从源点到 v 的最短路径的长度。假设从源点到 v 有一条更短的路径，如图 12.9 显示，这条路径先从 S 出发到某个顶点 x，然后继续到 v（甚至可能沿路重新进入 S）。如果这条路径比到 v 的着色路径更短，则从源点到 x 的初始路段也更短，于是有 distance[x]< distance[v]，因此贪心策略将选择 x 而不是 v 作为下一个加入 S 的顶点。

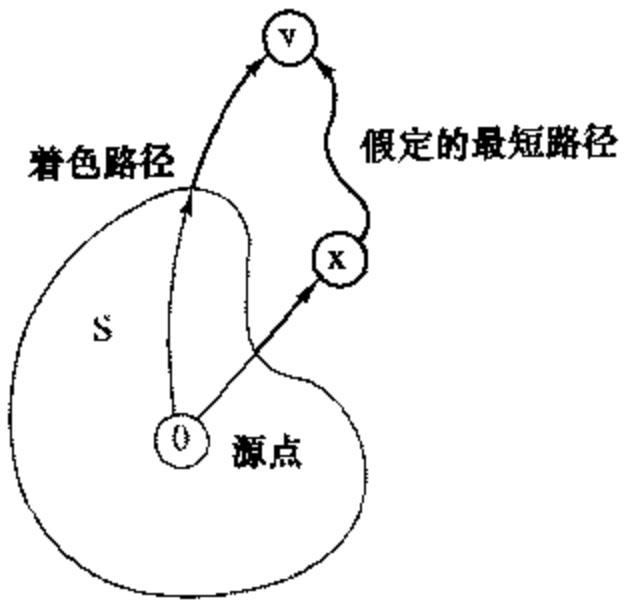


图 12.9　找到一条最短路径

＊＊＊证明结束

＊＊＊维持不变量

当将 v 加到 S 时，我们认为 v 现在已被着色（浅灰色），同时从源点到 v 的最短路径（它的所有边除最后一条外实际都已着色）也已着色。接下来，必须对不在 S 中的每个顶点 w，

检查通过 v 然后直接到 w 的一条路径是否比先前记录的到 w 的距离更短，从而更新整个 distance。也即，如果后者的数量更小，我们就用 distance[v]加上从 v 到 w 的边的权代替 distance[w]。

12.5.3 例子

在编写并入此方法的正式函数之前，我们先完成图 12.10 中显示的例子。对(a)部分显示的有向图，其初始情况显示在(b)中：集合 S(浅灰色的顶点)仅包含源点，即顶点 0，而距离表格 distance 的元素是在其他顶点旁以浅灰色显示的数值。到顶点 4 的距离最短，因此将 4

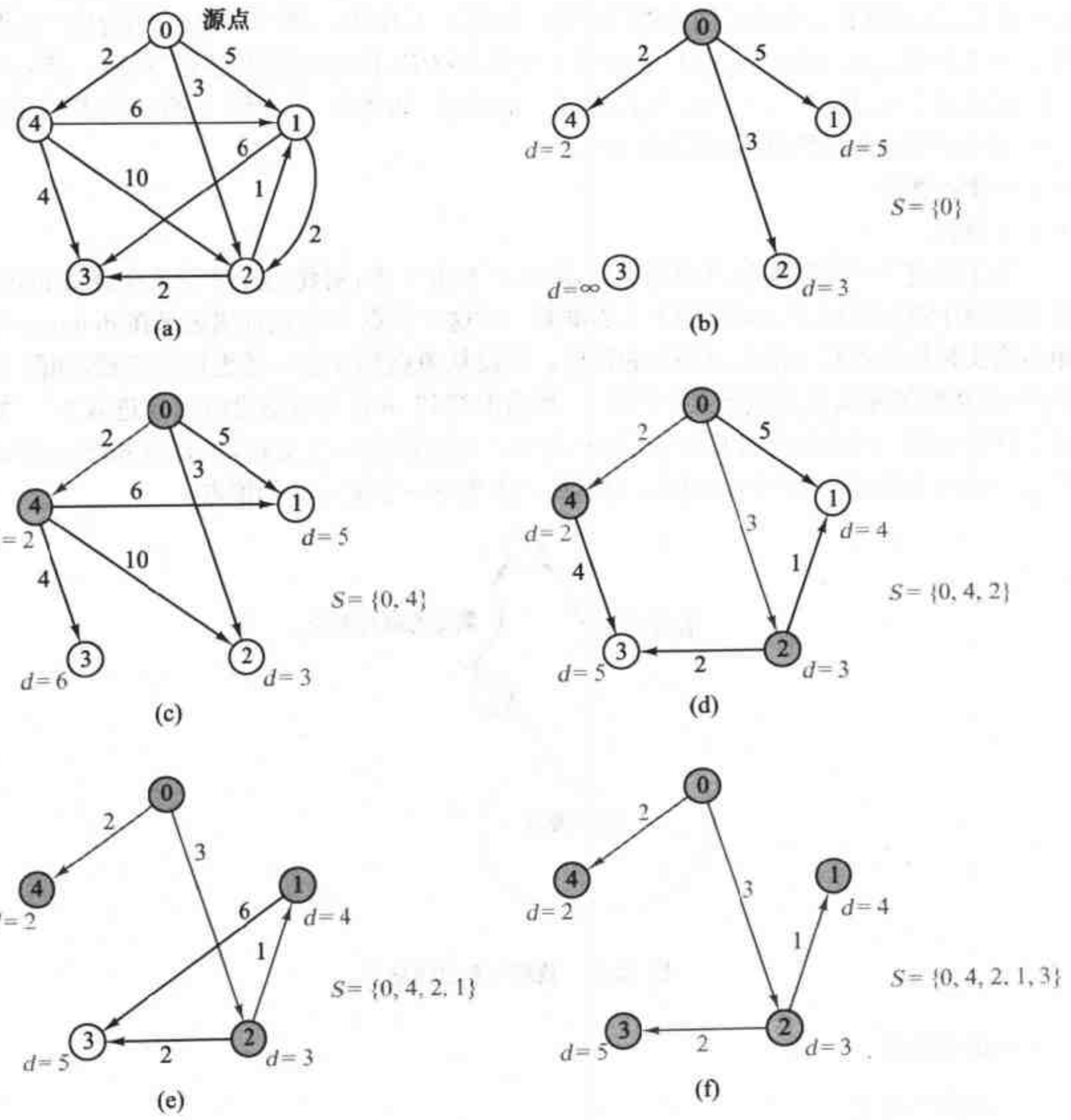


图 12.10 最短路径的例子

加入到(c)部分的 S 中，且距离 distance[3]的值更新为 6。因为经过顶点 4 到顶点 1 和 2 的距离比那些已在 T 中记录的值更大，因此它们的元素保持不变。到源点的下一个最近顶点是顶点 2，于是将它加入(d)部分，(d)也显示了到顶点 1 和 3 的距离更新后的效果，它们经过顶点 2 的路径比那些先前记录的路径更短。显示在(e)和(f)部分中的最后两步将顶点 1 和 3 加入 S，因此产生了在最后的图中显示的路径和距离。

12.5.4　实现

为了编写表达这个查找最短距离的算法的函数，我们必须选择有向图的一种实现。使用邻接表格实现可以方便随机访问图中的所有顶点，这一点正是我们在解决此问题时需要的。此外，通过把权存放在表格中，我们能用表格给出权和邻接关系。在下面的 Digraph 说明中，我们加入一个模板参数以允许客户程序说明要使用的权的类型。例如，使用 Digraph 模拟航空路线网的客户程序可能希望对航空路线的成本使用整数或实数。

```
template< class Weight, int graph_size>
class Digraph {
public:
  //  Add a constructor and methods for Digraph input and output.
  void set_distances(Vertex source, Weight distance[]) const;
protected:
  int count;
  Weight adjacency[graph_size][graph_size];
};
```

数据成员 count 记录了特定 Digraph 中的顶点数。在应用中，我们需要在这个类中加入用于输入和输出的方法，但因为这些方法在计算最短路径的 set_distances 方法的实现中用不上，因此我们将这些附加的方法留作练习。

我们假设 Weight 类有比较运算符。而且，我们期望客户程序定义称为 infinity 的最大可能的 Weight 值。比如，在整数权上操作的客户程序能使用 ANSI C++标准库<limits>中的信息并使用全局定义：**const** Weight infinity = numeric_limits < **int** > :: max();。我们将 infinity 的值放到相应的不存在边的邻接中。现在编写的方法 set_distances 将最近的距离表格算入它的输出参数 distance[]中。

* * * 最短距离过程

```
template< class Weight, int graph_size>
void Digraph<Weight, graph_size> :: set_distances(Vertex source,
                                    Weight distance[]) const
/*Post: The array distance gives the minimal path weight from vertex source to each
        vertex of the Digraph.*/
{
  Vertex v, w;
  bool found[graph_size]; //  Vertices found in S
  for(v=0; v<count; v++){
    found[v]=false;
    distance[v]=adjacency[source][v];
  }
  found[source]=true;    //  Initialize with vertex source alone in the set S.
```

```
  distance[source]=0;
  for(int i=0; i<count; i++){ //   Add one vertexv to S on each pass.
    Weight min=infinity;
    for(w=0; w<count; w++)if(! found[w])
      if (distance[w]<min){
        v=w;
        min=distance[w];
      }
    found[v]=true;
    for(w=0; w<count; w++)if(! found[w])
      if(min+ adjacency[v][w]<distance[w])
        distance[w]=min+ adjacency[v][w];
    }
}
```

* * *性能

为了估计这个函数的运行时间，我们指出，主循环要执行 $n-1$ 次，这里 n 是顶点数，而且在主循环内有其他两个循环，其中每个循环都执行 $n-1$ 次，这些循环提供了 $(n-1)^2$ 次操作。循环外所执行的语句仅为 $O(n)$，因此算法的运行时间是 $O(n^2)$。

12.6 最小生成树

12.6.1 问题

上一节中的最短路径算法的应用没有改变网络和图，也没有改变有向网和有向图。例如，在图 12.11 中，我们举例说明应用它去查找从源点 0 到网络的其他顶点的最短路径（以浅灰色显示）的结果。

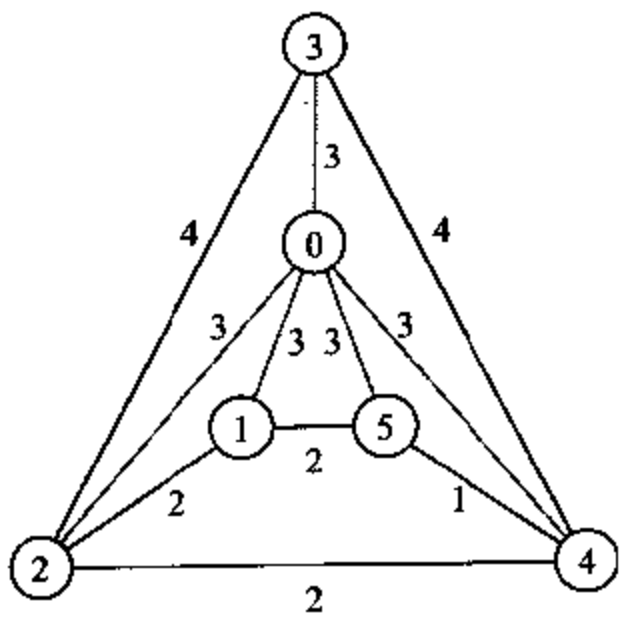


图 12.11 查找网络中的最短路径

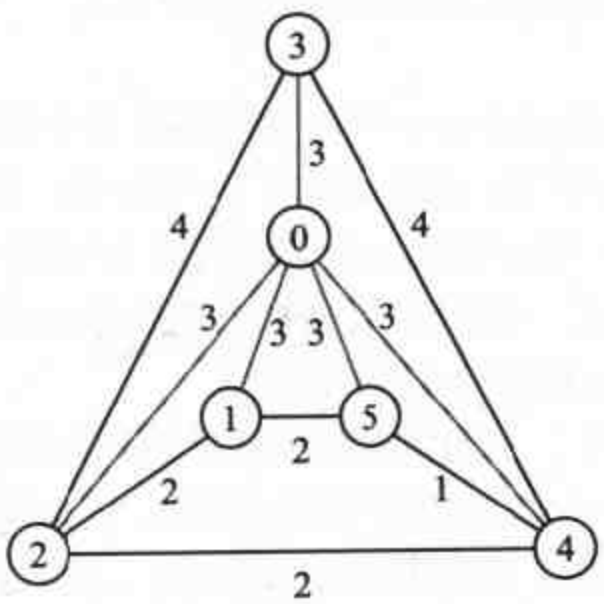
如果初始网络基于连通图 G，则从特定源点出发的最短路径就将这个源点和图 G 中所有其他的顶点连接起来了。因此，就如我们能在图 12.11 中所看到的，如果将计算过的最短路径合并到一起，我们得到一棵树，这棵树连接了 G 的所有顶点。换句话说，我们得到一棵连通树，它由 G 的所有顶点和一些边组成，我们称这样的树为 G 的生成树。如在前一节中，

我们把图 G 上的一个网络看作是航空路线图，每个顶点表示一个城市且每条边上的权表示从一个城市到另一个城市的飞行成本。G 的最小生成树表示路线的一个集合，这些路线将允许乘客完成城市间任何可能的旅行。当然，乘客将经常需要用几次飞行来完成旅行。然而，对于乘客来说，这种不方便性被较低的航线成本和较便宜的机票抵消了。事实上，最小生成树通常已被航线作为“中枢-辐射”(hub - spoke)路线系统采用。如果我们把图 12.11 想像成表示一个“中枢-辐射”系统，则源点对应于中枢机场，从这个顶点出现的路径是辐射路线。运行“中枢-辐射”系统的航线通过选择成本最小的路线系统去最小化它的成本这一点很重要。例如，图 12.12 中，网络的一对生成树用浅灰色边说明，航线将选择第二个生成树，因为它的标记和比较小。为了模拟一个优化的“中枢-辐射”系统，我们做出如下的定义：

* * * 最小生成树

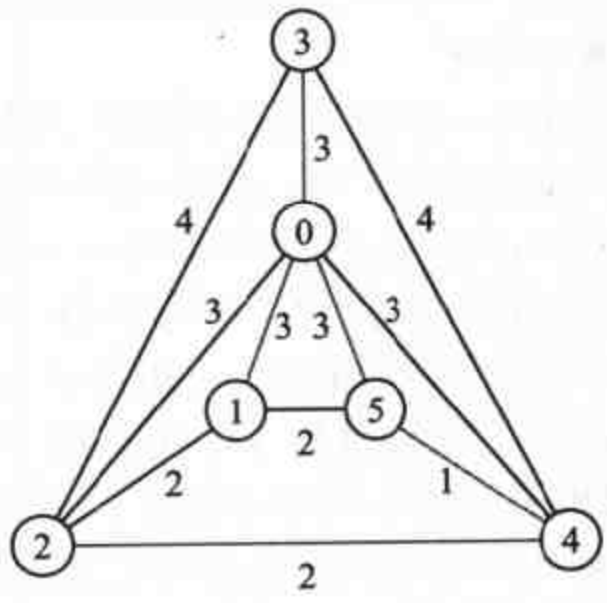
定义　连通网络的最小生成树是其边的权值总和尽可能小的生成树。

虽然并不难于比较图 12.12 中的两棵最小生成树，但却难以看出是否还有任何其他代价更小的生成树。我们的问题是设计一种确定连通网络最小生成树的方法。



树的权和=15

(a)



树的权和=12

(b)

图 12.12　网络中的两棵生成树

12.6.2　方法

因为最短路径算法能找出连通图的生成树，因此我们已经有了一个算法。它表明，能够对最短路径算法做出一个小小的改动，从而得到一种找出最小生成树的方法，这个方法最早由 R. C. Prim 在 1957 年实现。

我们选择一个称为源点的顶点开始，当我们用这个方法继续找出生成树时，我们保持一个顶点集 X，X 中的顶点到源点的路径可以在我们正在建立的最小生成树中找到。我们也需要跟踪边集 Y，Y 中的边连接了正在构造的树中的 X 的顶点。因此，经过一定的时间，可以想像 X 中的顶点和 Y 中的边构成了一棵小树，它生长成最终的生成树。最初，源点是 X 中惟一的顶点，边集 Y 为空。在每一步，我们都将另一个顶点加入 X：这个顶点的选择要使

回到 X 的一条边有尽可能小的权值，这条回到 X 的最小边要加入 Y。

对 Prim 算法给出一棵最小生成树的证明相当巧妙，我们将这个算法的证明推迟到这一节的最后。然而，通过认识到最终必须将连接 X 和网络中其他顶点的边合并到我们正在建立的生成树中，我们能理解加到 X 的新结点和加到 Y 的新边的选择。根据 Prim 准则选择的边提供了实现此连接的代价最小的办法，因此根据贪心准则，我们应该使用这条边。当我们开始实现 Prim 算法时，我们将维持一个属于 X 的顶点表，将它们作为布尔数组 component 的元素。我们可以方便地像存储图的边那样去存储 Y 中的边，这个图将通过我们的程序生长成一棵输出树。

* * * neighbor 表格

* * * distance 表格

我们将维持一张辅助表格 neighbor，对每个顶点 v，neighbor 给出了具有最小代价边的到 v 的 X 的顶点。维持另一张表格 distance 也会带来方便，distance 记录了这些最小代价。如果顶点 v 没有通过到 X 的边加入，则我们将它的距离记为无穷大值。在初始化 neighbor 表时，对所有的顶点 v，将 neighbor[v]设置成源点，在初始化 distance 时，如果从源点到 v 的边存在，我们将 distance[v]设置成从源点到 v 的边的权，否则设置为无穷大。

* * * 维持不变量

为了确定每一步要加入 X 的顶点，我们选择表 distance 中记录的具有最小值的顶点 v，这样的 v 尚未出现在 X 中。此后，我们必须更新这些表格以反映对 X 的修改。对每个不在 X 中的顶点 w，我们通过检查是否存在一条连接 v 和 w 的边来完成这个工作，如果这条边存在，则检查这条边的权是否比 distance[w]小。在有一条具有这种特征的边 (v,w) 的情况下，我们就将 neighbor[w] 重置为 v，并将 distance[w]重置为这条边的权。

例如，让我们完成图 12.13(a)部分显示的网络。初始情况显示在(b)部分：集合 X(浅灰色顶点)仅由源点组成，对每个顶点 w，顶点 neighbor[w]通过沿着图中从 w 出发的任意箭头显现。(distance[w]的值是相应边的权。)到顶点 1 的距离最短，因此将 1 加入到(c)部分的 X 中，并将表格 distance 中和 neighbor 中的元素更新为顶点 2 和 5，这些表中的其他元素保持不变。下一个到 X 的最近顶点是顶点 2，因此将它加入(d)部分，(d)也显示了 distance 和 neighbor 表格的更新后的效果。最后的 3 个步骤显示在(e)，(f)和(g)部分。

12.6.3 实现

为了实现 Prim 算法，首先必须选择表示网络的 C++类。这个算法与上一节的最短路径算法之间的相似性暗示了 Network 类应基于前面的 Digraph 类。

```
template< class Weight, int graph_size>
class Network:public Digraph<Weight, graph_size>{
public:
  Network();
  void read();                    //  overridden method to enter a Network
  void make_empty(int size=0);
  void add_edge(Vertex v, Vertex w, Weight x);
  void minimal_spanning(Vertex source,
                        Network<Weight, graph_size>&tree) const;
};
```

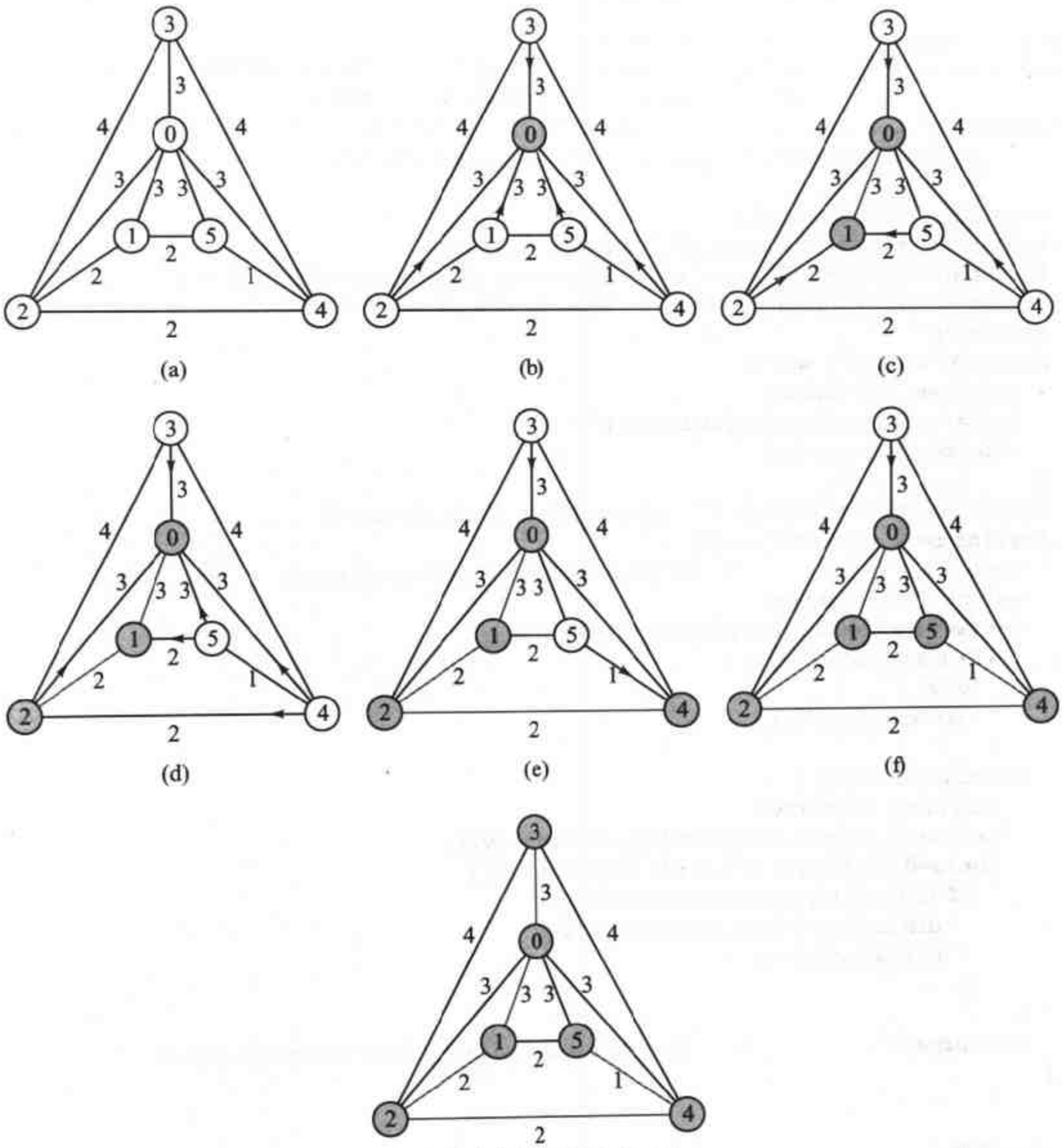


图 12.13　Prim 算法的例子

我们在这里重载了输入方法 read，以确信任意边(v,w)的权匹配边(w,v)的权，这样可以保护数据结构不受无向边潜在的讹误的影响。新方法 make_empty(**int** size)创建了一个具有 size 个顶点但没有边的 Network。另一个新方法 add_edge 将一条具有指定权的边加到 Network。就像上一节一样，我们假设 Weight 类有比较运算符。此外，我们希望客户程序去定义最大可能的称为无穷大的 Weight 值。我们现在编写的方法 minimal_spanning 将计算最小的生成树并将它作为自己的输出参数 tree。虽然，当将此方法运用到连通的网络

Network 时，它仅计算生成树，但它总是计算 Network 中由源点确定的连通成分的生成树。

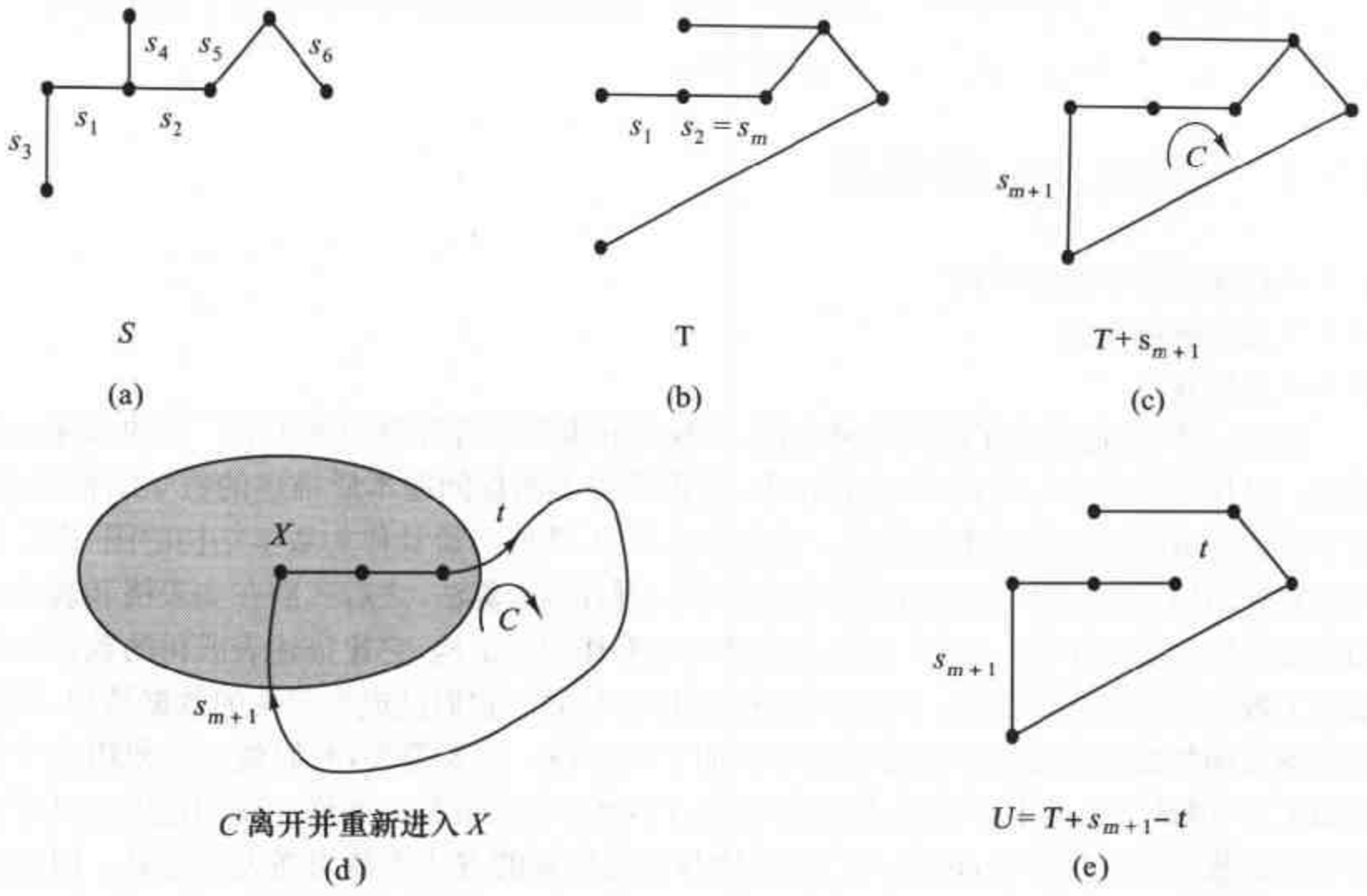
```
template<class Weight, int graph size>
void Network<Weight, graph_size> <::minimal_spanning(Vertex source,
                        Network<Weight, graph_size>&tree) const
/*Post: The Network tree contains a minimal spanning tree for the connected component of
        the original Network that contains vertex source. */
{
  tree.make_empty(count);
  bool component[graph_size];//   Vertices in set X
  Weight distance[graph_size];//   Distances of vertices adjacent to X
  Vertex neighbor[graph_size];//   Nearest neighbor in set X
  Vertex w;
  for(w=0; w<count; w++){
    component[w]=false;
    distance[w]=adjacency[source][w];
    neighbor[w]=source;
  }
  component[source]=true; //   source alone is in the set X.
  for(int i=1; i<count; i++){
    Vertex v;                 //  Add one vertexv to X on each pass.
    Weight min=infinity;
    for(w=0; w<count; w++)if(!component[w])
      if(distance[w]<min){
        v=w;
        min=distance[w];
      }
    if(min<infinity){
      component[v]=true;
      tree.add_edge(v,neighbor[v],distance[v]);
      for(w=0; w<count; w++)if(!component[w])
        if (adjacency[v][w]<distance[w]){
          distance[w]=adjacency[v][w];
          neighbor[w]=v;
        }
    }
    else break;               //   finished a component in disconnected graph
  }
}
```

* * *性能

为了估计这个函数的运行时间，我们注意到循环执行了 $n-1$ 次，其中 n 是顶点数，在主循环内还有其他两个循环，其中每个循环各执行 $n-1$ 次，因此这些循环构成了复合的 $(n-1)^2$ 次操作。循环外的语句仅构成 $O(n)$，因此整个算法的运行时间是 $O(n^2)$。

12.6.4 Prim 算法的验证

必须证明，对连通图 G，由 Prim 算法产生的生成树 S，其边权的总和小于 G 的任何其他生成树的边权的总和。Prim 算法确定了一个由边 $s_1, s_2, \ldots, s_n$ 构成的序列，它们构成了树 S。如图 12.14 所示，这儿 s_1 是 Prim 算法中加入集合 Y 的第一条边，s_2 是加入 Y 的第二条边，以此类推。



S　　T　　$T + s_{m+1}$

(a)　　(b)　　(c)

C 离开并重新进入 X　　$U = T + s_{m+1} - t$

(d)　　(e)

图 12.14　Prim 算法的输出树的最佳性

* * * 归纳：基本情况

* * * 最终情况

为了证明 S 是最小生成树，我们改为证明：如果 m 是一个整数且 $0 \leqslant m \leqslant n$，若 $i \leqslant m$，则有一棵包含了边 s_i 的最小生成树。我们通过对 m 进行归纳来证明这个结果。基本情况是 $m = 0$，此时当然为真，因为任何最小生成树确实包含了必不可少的空的边集。而且一旦完成归纳，$m = n$ 的最终情况表明有一棵包含 S 的所有边的最小生成树，因此它与 S 一致。（注意将任意一条边加到生成树会产生一个环，因此任何包含 S 中所有边的生成树一定是 S 本身。）换句话说，一旦完成归纳，则已表明 S 是最小生成树。

* * * 归纳步骤

因此必须建立归纳步骤，证明如果 $m < n$ 且 T 是包含边 $s_i (i \leqslant m)$ 的最小生成树，则存在一棵最小生成树 U，它具有这些边和 s_{m+1}。如果 s_{m+1} 已属于 T，则只要简单地设 $U = T$，因此我们假设 s_{m+1} 不是 T 的边。见图 12.14(b)部分。

设属于边 $s_1, s_2, \cdots\cdots, s_m$ 的 S 的顶点集记作 X，S 的剩余顶点集记作 R。我们记得在 Prim 算法中，选出的边 s_{m+1} 连接了 X 的一个顶点和 R 的一个顶点，且 s_{m+1} 至少和这两个集合之间的某条其他的边具有同样的代价。考虑将 s_{m+1} 加入 T 的结果，如图 12.14(c)部分说明。这条边的加入一定会产生环 C，因为连通网络 T 一定包含了连接 s_{m+1} 端点的多边路径。环 C 一定包含连接 X 和 R 的边 $t \neq s_{m+1}$，因为沿着闭合路径移动时，进入集合 X 的次数和离开 X 的次数恰好是一样的，见图 12.14(d)部分。Prim 算法保证 s_{m+1} 的权小于或等于 t 的权。因此，通过从 T 删除 t，并加入 s_{m+1} 得到新的生成树 U（见图 12.14(e)部分），U 的权和不

大于 T 的权和。我们推断 U 一定也是 G 的一棵最小生成树，但 U 包含了边的序列 s_1，s_2，……，s_m，s_{m+1}。因此完成归纳。（**证明结束**）

12.7 图作为数据结构

* * * 数学结构和数据结构

* * * 灵活性和能力

* * * 无规律性

在这一章，我们研究了图的几种应用，但我们并未触及有关图的算法的广泛的和深层的主题。就像在本章中一样，在许多算法中，图都是作为捕捉问题本质描述的数学结构出现，而不是作为解决它的计算工具出现。注意在本章中，我们已谈及作为数学结构的图，而不是作为数据结构，因为我们已经用图去表达数学问题并编写算法，之后我们在如表格和表这样的数据结构上实现了图。然而，图本身当然可以看作数据结构，它比描述表或树的数据结构表达了数据间更复杂的关系。因为图的通用性和灵活性，它们已成为强大的数据结构，在更高级的应用如数据库管理系统的设计中证明了其价值。只要需要，我们就应当使用这个强大的工具，但必须总是小心使用，使它们的能力不至于变成混乱。或许在使用强大工具时最安全的措施是坚持规律性；即仅以仔细设计和完善理解的方式去使用强大的工具。因为图的通用性，因此将这个要求强加到它们的使用中并不总是一件简单的事情。虽然如此，在这个世界中，不管我们如何努力去避免无规律性，它们总会逐渐产生。系统分析员和程序设计者在努力维护基本的系统设计的完整性的同时，对这些无规律性的适应将是致命的。无规律性甚至出现在我们设计用作数据结构模型的系统中，如家族树模型，我们已经使用了它的术语。下面这个经典的故事极好地说明了会发生什么，N. Wirth[①] 曾在 1922 年 7 月某份苏黎世报纸中引用它。

我和一个寡妇结婚了，她有一个已长大的女儿。我父亲经常去拜访我们，他与我的继女相爱并结婚了。因此，我的父亲成了我的女婿，我的继女也成了我的母亲。几个月后，我的妻子生了个儿子，他也成了我父亲的内弟，也成了我的叔叔。我父亲的妻子，也就是我的继女，也有一个儿子。因此，我有了一个弟弟，同时他也是外孙。我的妻子是我的祖母，因为她是我母亲的母亲。因此，我是我妻子的丈夫，同时也是她的继孙，换句话说，我是我自己的祖父。

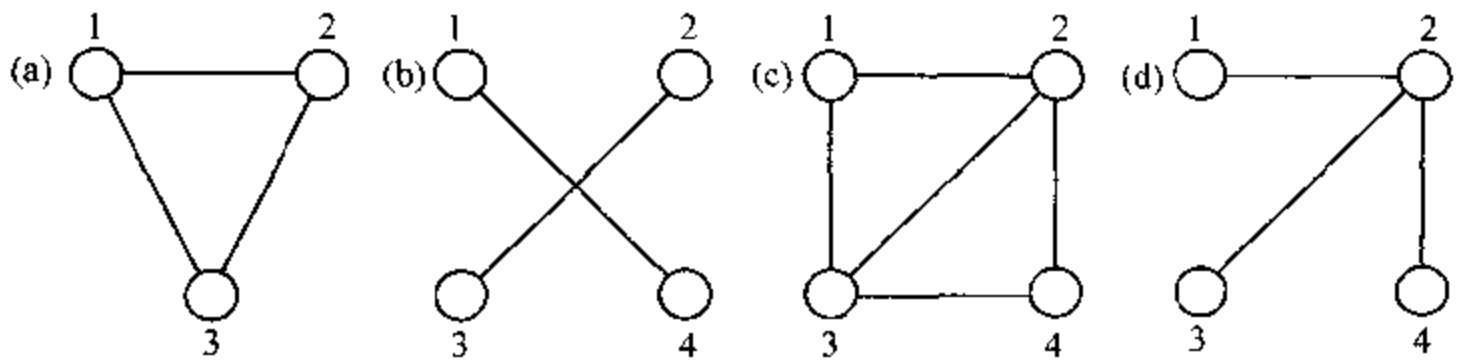
练习 12.7

E1. (1)找出下列每个图(见下页)中所有的环。(2)这些图中哪一些是连通的？(3)这些图中哪一些是自由树？

E2. 对练习 E1 中所示的每个图，给出它的实现，以(1)邻接表格；(2)具有链式邻接表的链式顶点表；(3)顺序邻接表的顺序顶点表。

E3. 如果图的每个顶点具有相同的价(也即，如果每个顶点都邻接相同数目的其他顶点)，则

① Algorithms+Data Structures=Programs, Prentice Hall, Englewood Cliffs, N.J., 1976, page 170.



这个图是规则的。对规则的图，一种好的实现是使链表和邻接表中的顶点保持连续。所有邻接表的长度称为这个图的度。为规则图的这种实现编写 C++类说明。

E4. 课文中给出的拓扑排序函数在检错方面有所不足。修改(1)深度优先和(2)广度优先函数使得它们将检测图中任何(有向)环，以及指示什么顶点因为落在环上而不能放在任何拓扑次序中。

E5. 我们如何确定网络中的最大生成树？

E6. 克鲁斯卡尔(Kruskal)算法通过以递增权的次序考虑所有的边，从而计算网络中的最小生成树。在将边加入初始为空的集合的过程中，为生成树选择边。在边的选择中，如果一条边和先前选择的边一道没有产生环，则这条边被选中。证明克鲁斯卡尔算法选择的边确实形成连通网络的一棵最小生成树。

E7. 迪杰思特拉(Dijkstra)算法通过以任何方便的次序考虑所有的边，从而计算网络中的最小生成树。就像在克鲁斯卡尔算法中一样，通过将边加入初始为空的集合中，我们选择生成树的边。然而，现在在考虑每条边时，如果它和先前选择的边一道产生了环，则选中这条边，并将这个环中最高价的边取消选中。证明由迪杰思特拉算法选择的边也形成连通网络的最小生成树。

程序设计项目 12.7

P1. 编写 Digraph 类的方法 read，它从终端读取无向图的顶点数和邻接顶点表。确信包含错误检查。图用下列方式实现：

(1)邻接表格；

(2)具有链式邻接表的链式顶点表；

(3)具有链式邻接表的顺序顶点表。

P2. 编写 Digraph 类的方法 write，它向终端输出图的相关信息说明。图用下列方式实现：

(1)邻接表格；

(2)具有链式邻接表的链式顶点表；

(3)具有链式邻接表的顺序顶点表。

P3. 使用方法 read 和 write 去实现和测试这一节设计的拓扑排序函数：

(1)深度优先次序

(2)广度优先次序

P4. 编写 Digraph 类的方法 read 和 write，它们将为 12.5 节的实现执行输入和输出。确信方法 write()也可以应用于 12.6 节的派生类 Network。

P5. 实现并测试带权有向图中确定最短距离的方法。

P6. 实现并测试确定连通网络最小生成树的 Prim 算法、克鲁斯卡尔算法和迪杰思特拉算法方法。

启示和易犯的错误

1. 图提供了描述许多应用的本质特征的极好方法，因而便于说明基本问题和明确地表达求解问题的算法。图有时以数据结构出现，但更多的时候是以有助于问题求解的数学抽象出现。
2. 可以使用不同种类的数据结构以多种方法实现图。延缓图的实现选择，直到在问题求解和算法设计阶段图的应用得到了充分的理解。
3. 许多应用都需要图的遍历。让应用决定遍历的方法：深度优先、广度优先或采用某种其他次序。深度优先遍历自然是递归的(或能够使用栈)。广度优先遍历通常使用队列。
4. 贪心算法仅仅表示在设计图算法时有用的许多范例的一个样本。更多的方法和例子，查阅参考书目。

复习题

12.1

1. 就本章来说，什么是图？什么是边和顶点？
2. 无向图和有向图之间的区别是什么？
3. 定义术语邻接、路径、环和连通。
4. 对有向图，强连通是何含义？弱连通呢？

12.2

5. 描述在计算机内存中实现图的 3 种方法。

12.3

6. 解释图的深度优先遍历和广度优先遍历之间的区别。
7. 在(1)深度优先和(2)广度优先遍历期间，需要什么数据结构去记录跟踪等待的顶点？

12.4

8. 拓扑排序定义哪几种图？
9. 对这样的图，拓扑次序是什么？

12.5

10. 为什么找到最短距离的算法称为贪心的？

12.6

11. 解释找出最小生成树的 Prim 算法和克鲁斯卡尔算法如何不同？

进阶参考书目

研究图和处理它们的算法是一个较大的主题，这个主题同时涉及数学和计算机科学。

有 3 本书都包含了许多有趣的算法，它们是：R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983, 131 pages. Shimon Even, *Graph Algorithms*, Computer Science Press, Rockville, Md., 1979, 249 pages.

E. M. Reingold, J. Nievergelt, N. Deo, *Combinatorial Algorithms: Theory and Practice*, Prentice - Hall, Englewood Cliffs, N. J., 1977, 433 pages.

决定图的最短路径的贪心算法的原始参考是：E. W. Djkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik* 1(1959), 269 - 271.

找到最小生成树的 Prim 算法的报道是：R. C. Prim, "Shortest connection networks and some generalizations", *Bell System Technical Journal* 36(1957), 1389 - 1401.

克鲁斯卡尔算法的描述见：J. B. Kruskal, "On the shortest spanning tree of a graph and the traveling salesman problem", *Proceedings of the American Mathematical Society* 7 (1956), 48 - 50.

找出最小生成树的迪杰思特拉算法的原始参考是：E. W. Dijkstra, "Some theorems on spanning subtrees of a graph", *Indagationes Mathematicæ* 28(1960), 196 - 199.

第 13 章　案例研究:波兰表示法

本章研究用于算术或逻辑表达式的波兰表示法,我们首先在问题求解方面进行研究,然后再将它应用到一个程序中,这个程序交互式地接受一个表达式,编译这个表达式并求出它的值。本章说明递归、栈、表格和树的用法,以及它们在问题求解和算法设计中的相互影响。

13.1　问题

* * * 词源:Fortran

计算机语言的早期设计者最重要的成就之一是允许程序设计者能采用非常接近于通常的数学形式去书写算术表达式。设计能理解如下表达式并生成机器语言输出的编译器是一个真正的成功:

$$(x+y)*exp(x-z)-4.0$$

$$a*b+c/d-c^{*}(x+y)$$

$$!(p\ \&\&\ q)\ \|\ (x<=7.0)$$

实际上,Fortran 这个名字就是因为这种非凡的成就而被公认为是代表 Formula Translator。既然是这样,将表达式翻译成机器语言的指令序列经常只是采用一种简单的思想,而这种思想一旦得到完全理解,则会为困难问题的完美解答提供线索。

与人们可能采用的第一种方法形成对照,在本章中要展开介绍的这种方法的成功之处在于它在翻译表达式时不必重复扫描它,而且在预翻译后,也不必考虑括号和运算符的优先级,从而可以非常高效地实现表达式的求值。

13.1.1　二次公式

* * * 符号

在讨论这一思想之前,我们先简要地想像一下早期的编译器设计者在遇到一个非常复杂的表达式时可能面临的问题。即使是二次公式也会带来问题:

$$x=(-b+(b\uparrow 2-(4\times a\times c)\uparrow \frac{1}{2})/(2\times a)$$

在这里以及本章整个篇幅中,我们用'↑'表示取幂。当然在 C++中并不存在这个运算符,但我们在本章将设计自己的表达式系统,而且可以按照我们希望的任何方式自由设立自己的约定。当我们取平方根时,将采用$\uparrow \frac{1}{2}$,并且仅将注意力限制在非负根上。

哪一种运算符应在其他运算符之前执行?括号的作用是什么?何时可以省略括号?当我们面对这个例子回答这些问题时,可能需要前后几次查看这个表达式。

在考虑如何翻译这样的表达式时,编译器设计者很快停留在我们现在熟悉的惯例上:通常是从左到右进行运算,服从运算符的优先级,其中指数最高,然后是乘法和除法,再后是加

法和减法，这种优先次序可以用括号去改变。在这个二次公式中，运算符的优先次序是：

$$x=(-b+(b\uparrow 2-(4\times a\times c)\uparrow \frac{1}{2})/(2\times a)$$

$$\begin{array}{cccccccccc} \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \\ 10 & 1 & 7 & 2 & 5 & 3 & 4 & 6 & 9 & 8 \end{array}$$

注意赋值号=实际上是一个运算符，它取出自己右边的操作数的值，并赋给左边的操作数，=的优先级是所有运算符中最低的，因为在完全求出表达式的值之前将无法执行这个运算符。

1. 一元运算符和优先级

除了一种运算符例外，这个二次等式中的所有运算符都是二元的，即它们有两个操作数。这个例外是 $-b$ 中的前导减号，它是个一元运算符，在确定优先级时一元运算符带来了一点复杂性。通常我们将 -2^2 解释成 -4，它意味着负号是在求幂之后。但是我们将 2^{-2} 解释成 $\frac{1}{4}$ 而不是 -4，因此在这里是先求负号。我们给一元运算符赋予与求幂运算同样的优先级是合理的，并且为了维持通常的代数惯例，我们将从右到左求这种优先级的运算符的值。而且，这样做也给出了 $2\uparrow 3\uparrow 2$ 的通常的解释，它被解释成 $2^{(3^2)}=512$，而不是 $(2^3)^2=64$。

除负号以外还有其他的一元运算符，它们包括：x 的阶乘表示为 $x!$，函数 f 的导数表示为 f'，以及所有单个变量的函数，如三角函数、指数函数和对数函数等。在 C++ 中，也有一个布尔(Boolean)运算符!，它对一个布尔变量求反。为避免与阶乘运算符混淆，我们在本章设计的表达式系统中用'not'表示布尔取反。

一些二元运算符也有布尔结果：运算符 && 和 ||，以及比较运算符==，!=，<，>，≤和≥。这些比较通常在算术运算符之后，但在 &&，|| 和赋值运算符之前执行。

*** 优先级列表

我们因此得到下面的优先级列表，它反映了我们在求运算符的值时通常的习惯。

运算符	优先级
↑，任何一元运算符	6
×，%	5
+-(二元)	4
== != < > ≤ ≥	3
Not	2
&& \|\|	1
=	0

*** 运算符的优先级比较

注意这个表格中显示的优先级与 C++ 中使用的那些优先级并不是一样的，例如在 C++ 语法规则中，运算符! 具有与一元运算符同样的优先级，这意味着在 C++ 的表达式中必须经常使用括号。即使如此，给'not'分配一个较低的优先级，人工阅读表达式时也会感到比较

清晰。例如表达式 not $x<10$ 按 C++惯例将解释成$(!x)<10$，它始终为真。

13.2 思想

13.2.1 表达式树

画出图形经常是看透一个问题的极好的方法。对我们目前的问题，合适的图形是表达式树(expression tree)，它在 10.1.2 节首次引入。回忆一下，表达式树是一种二叉树，其中树叶是简单的操作数，而内部顶点是运算符。如果运算符是二元的，则它有两个非空的子树，分别是它的左、右操作数(或者是简单的操作数或者是子表达式)。如果运算符是一元的，则它只有一个子树是非空的，至于是左边或右边，要根据这个运算符是写在其操作数的右边还是左边来定。请复习图 10.3 中几个简单的表达式树，以及图 10.4 中二次公式的表达式树。

例如，我们确定如何求图 13.1 的(a)部分中显示的一个表达式树的值。很显然必须从其中一个树叶开始，因为它只是简单的操作数，我们一开始就知道它的值。为前后一致，我们从最左边的树叶开始，它的值是 2.9。在这个例子中，直接在这个树叶之上的运算符是一元取负运算，因此我们就能立即应用这个运算符并用结果 -2.9 去替换这个运算符及其操作数。这一步导致图的(b)部分中菱形结点。

(b)部分中菱形结点的双亲是一个二元运算符，这个运算符的另一个操作数尚未求出，因此目前还不能应用它，而是必须考虑用浅灰色路径显示的下两个树叶。在移过这两个树叶后，路径就移到了它们的双亲运算符，这时已能够求出它的值，并将结果放在(c)部分中显示的第二个菱形结点中。

在此阶段，加法的两个操作数均已可用，因此我们执行加法，获得(d)部分中简化的树，我们如此继续，直到这棵树被约简成单个结点，它就是最终的结果。总而言之，我们是以下面的次序处理树中的结点：

2.9 -2.7 3.0 /+5.5 2.5 -! ×

* * * 后序遍历

对任何以给定运算符为根的子树，通常我们应该按照下面的次序处理：

求出左子树的值；求出右子树的值；执行运算符。

(如果运算符是一元的，则这些步骤之一为空。)这种次序正是表达式树的后序遍历。我们已在 10.1.2 节注意到表达式树的后序遍历产生表达式的后缀式，其中每个运算符都写在它的操作数之后，而不是它们之间。

这一简单的思想是用计算机有效计算表达式的关键所在。

事实上，我们将运算符放在其操作数之间的这种书写算术或逻辑表达式的习惯方式有点不合逻辑，指令：

取数字 12 并乘上……

在给出第二个因数之前是不完整的。同时它必须同时记住一个数和一个操作。从建立统一规则的立场出发，下面这两种写法更有意义：

取数字 12 和 3；然后相乘。

或者

做一次乘法，数字是 12 和 3。

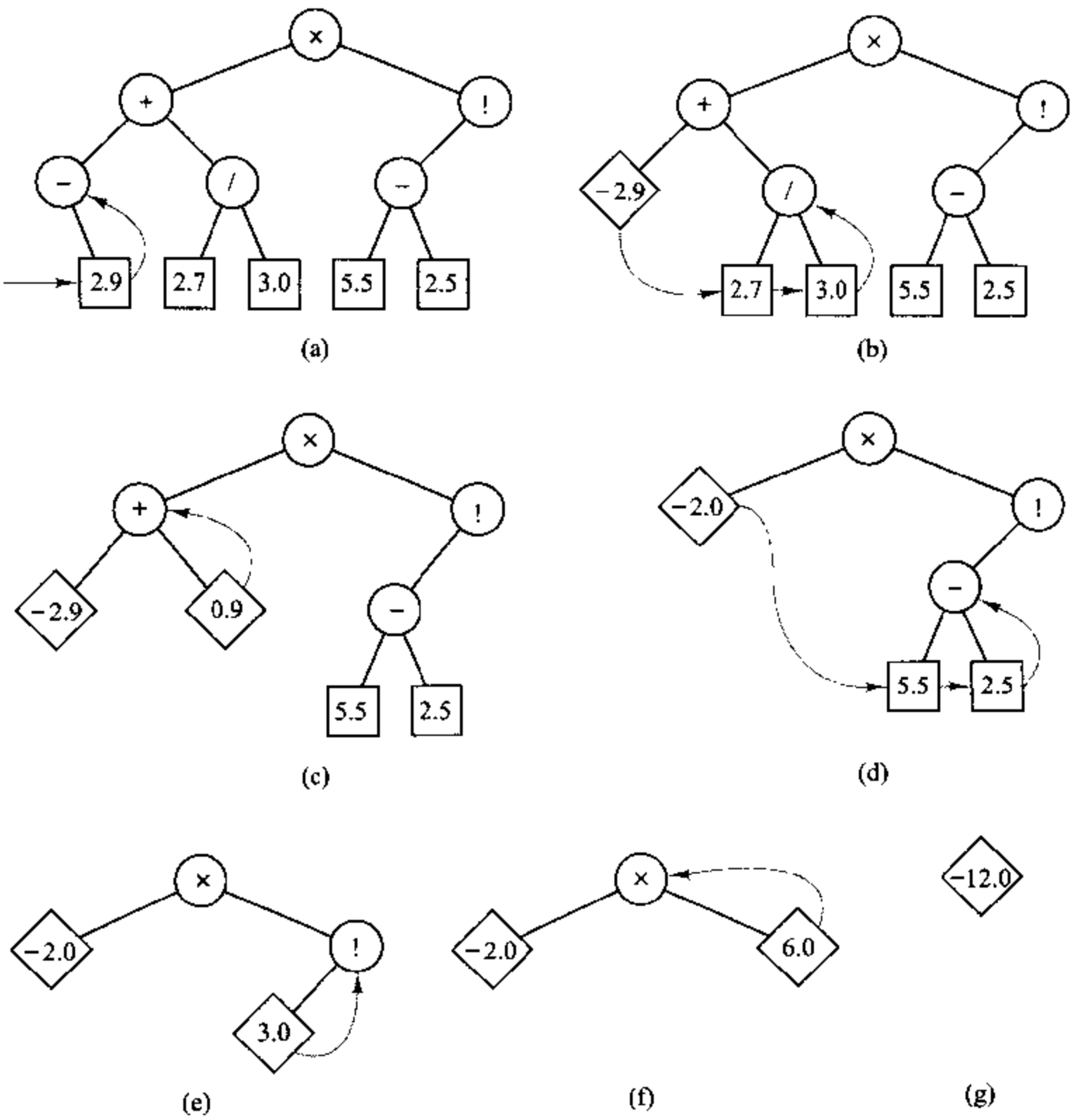


图 13.1　表达式树的求值

13.2.2　波兰表示法

这种将所有的运算符写在其操作数之前或之后的方法称为波兰表示法(Polish notation)，这是为纪念它的发明者、波兰数学家 Jan Łukasiewicz。当运算符写在其操作数之前时，称为前缀式(prefix form)；当运算符出现在其操作数之后时，称为后缀式(postfix form 或 suffix form)，或者有时也称为逆波兰式(reverse Polish form)。最后，在这个上下文中，习惯上使用杜撰的新短语 infix form(中缀式)表示将二元运算符写在其操作数之间。

表达式 $a \times b$ 在前缀表达式中变成 $\times ab$，在后缀式中变成 $ab\times$。在表达式 $a+b \times c$ 中，先做乘法，因此我们首先转换它，用后缀式先得到 $a+(bc\times\)$，然后再得到 $abc\times+$。这个表达

式的前缀式是$+a\times bc$。注意，前缀和后缀式并不能通过镜像或其他简单的变换建立关系，同时注意在波兰形式中已省略了所有的括号，我们将在后面证明这种省略的合理性。

作为一个更复杂的例子，我们能从图 10.4 中所示的那个二次公式的表达式树开始，写出它的前缀和后缀式。

＊＊＊前序遍历

首先，我们以前序遍历这棵树。树根中的运算符是赋值号‘=’，之后我们转到它的左子树，这棵左子树仅包含操作数 x。根的右子树从除号‘/’开始，然后向左转到‘+’和一元负号‘-’上。

现在我们有一处模棱两可的地方，如果不加以改正的话，则会后患无穷。表达式中的第一个‘-’(减号)是一元取反，第二个是二元减法，在波兰形式中弄不清谁是谁。当我们求前缀字符串的值时，我们不知道是为‘-’取一个还是两个操作数，得到的结果也有很大的差别。为避免这种模棱两可的情况，在这一章我们通常保留‘-’表示二元减法，而将特殊的符号‘~’用于一元取负。(这个符号当然并不标准，也有解决这个问题的其他方法。)

至此，图 10.4 的前序遍历产生：

$$=x/+\sim b$$

下一步是遍历运算符‘+’的右子树，结果得到序列：

$$\uparrow - \uparrow b\ 2 \times \times 4\ a\ c\ \frac{1}{2}$$

最后，我们遍历除号‘/’的右子树，得到：

$$\times\ 2\ a$$

因此这个二次公式完整的前缀式是：

$$=x/+\sim b\ \uparrow - \uparrow b\ 2 \times \times 4\ a\ c\ \frac{1}{2} \times 2\ a$$

可以验证它的后缀式是：

$$x\ b\sim b\ 2\ \uparrow\ 4\ a \times c \times -\frac{1}{2}\ \uparrow +2\ a \times /=$$

练习 13.2

(1)画出下列每个表达式的表达式树，使用这些树，将表达式转换成(2)前缀式和(3)后缀式，使用这一节而不是 C++中设计的优先级表。

E1. $a+b<c$

E2. $a<b+c$

E3. $a-b<c-d \| e<f$

E4. $n!/(k!\times(n-k)!)$ (二项式系数公式)

E5. $s=(n/2)\times(2\times a+(n-1)\times d)$(这是算术级数前 n 项的和)

E6. $g=a\times(1-r^n)/(1-r)$(几何级数前 n 项的和)

E7. $a==1 \| b\times c==2 \| (a><1$ && not $b<3)$

13.3　波兰表达式的求值

我们首先引入前缀式，作为遍历表达式树的一种自然的顺序，用于求出相应表达式的值。这一节稍后，我们将设计算法从后缀式直接求出表达式的值，但首先（因为它比较简单）我们将考虑前缀式。

13.3.1　对前缀形式的表达式求值

二叉树的前序遍历是自顶向下进行，首先是访问树根，之后余下的遍历将分成两个部分。组织这个过程的自然方式是将这个过程作为一个递归的、分而治之的算法。同样的情况对前缀形式的表达式也成立。第一个符号（如果有多于一个符号的话）是一个运算符（实际上是最后执行的那一个），表达式的其余部分包括此运算符的操作数（对一元运算符有一个，对二元运算符有两个）。我们用于对前缀式求值的函数因此从这第一个符号开始。如果它是一个一元运算符，则此函数递归地调用自己去确定操作数的值；如果第一个符号是一个二元运算符，则函数两次递归调用求出它的两个操作数。递归在剩余情况下终止：当第一个符号是简单的操作数，则它是自己的前缀式，函数因此应仅返回它的值。当然，如果没有第一个符号，则我们必须产生一个 Error_code。

下面的函数概要总结了求解前缀形式表达式的值的方法：

```
* * * 概要

Error_code Expression :: evaluate_prefix(Value &result)
/* Outline of a method to perform prefix evaluation of an Expression. The details depend
   on further decisions about the implementation of expressions and values. */
{
  if (the Expression is empty) return fail;
  else {
    remove the first symbol from the Expression, and
      store the value of the symbol as t;
    if (t is a unary operation) {
      Value the_argument;
      if (evaluate_prefix(the_argument) == fail) return fail;
      else result = the value of operation t applied to the_argument;
    }
    else if (t is a binary operation) {
      Value first_argument, second_argument;
      if (evaluate_prefix(first_argument) == fail) return fail;
      if (evaluate_prefix(second_argument) == fail) return fail;
      result = the value of operation t
                 applied to first_argument and second_argument;
    }
    else                          //  t is a numerical operand.
      result = the value of t;
  }
  return success;
}
```

13.3.2 C++约定

为了约束这个概要中的细节，需要建立一些约定并重新用C++编写这个算法。我们用下面这个类的增大版本去表示表达式。

```
class Expression {
public:
  Error_code evaluate_prefix(Value &result);
  Error_code get_token(Token &result);
  //  Add other methods.
private:
  //  Add data members to store an expression.
};
```

表达式中的运算符和操作数可能有超过一个字符长的名字，因此我们不是一次扫描表达式的一个字符，而是对表达式中的单个运算符和操作数定义一个标记。在我们的程序中，将标记表示为结构 Token 的对象。

为了强调求值方法只用从头到尾扫描表达式一次，我们要应用另一个方法 Token Expression::get_token();。这个函数将从头到尾移过表达式，每次删除并返回一个标记。我们需要知道这个标记是否是操作数、一元运算符还是二元运算符，因此我们假设存在Token 类的方法 kind()，它将返回下面这个枚举类型的一个值：

```
enum Token_type {
  operand, unaryop, binaryop   //  Add any other legitimate token types.
};
```

为简单起见，我们假设所有的操作数和操作符的求值结果具有同样的类型，这种类型我们现在不予指定，仅称它为 Value，在许多应用中，这种类型将是 int，double 或 bool 中的一种。

最后，必须假设存在 3 个辅助函数，它们返回 Value类型的结果。前两个函数是：

```
Value do_unary(const Token &operation, const Value &the_argument);
Value do_binary(const Token &operation,
                const Value &first_argument, const Value &second_argument);
```

这两个函数实际上在操作数上执行给定的操作，它们需要识别用于此操作的符号和自变量，并调用必要的机器语言指令。类似地，Value get_value(const Token &operand);这个函数返回数字操作数的实际值，而且可能需要，例如将一个常数从十进制转换成二进制形式或者查看一个变量的值。这些函数的实际格式在很大程度上依赖于应用，我们无法在这里解决所有的问题，而只是想将重点放在设计编译器或表达式计算器的一个重要的部分。

13.3.3 前缀求值的 C++函数

有了这些预备措施，我们现在能指定算法概要中更多的细节，将它转换成对前缀表达式求值的 C++方法。

```
Error_code Expression::evaluate_prefix(Value &result)
/*Post: If the Expression does not begin with a legal prefix expression, a code of fail is
        returned. Otherwise a code of success is returned, and the Expression is
        evaluated, giving the Value result. The initial tokens that are evaluated are
        removed from the Expression.*/
{
  Token t;
  Value the_argument, first_argument, second_argument;
  if(get_token(t)==fail) return fail;
  switch(t.kind()) {
  case unaryop:
    if(evaluate_prefix(the_argument)==fail) return fail;
    else result=do_unary(t, the_argument);
    break;
  case binaryop:
    if(evaluate_prefix(first_argument)==fail) return fail;
    if(evaluate_prefix(second_argument)==fail) return fail;
    else result=do_binary(t, first_argument, second_argument);
    break;
  case operand:
    result=get_value(t);
    break;
  }
  return success;
}
```

13.3.4　后缀表达式的求值

* * * 与前缀求值的比较

由于后缀式实际上是代数表达式的一种“自顶向下的”公式表示,因此在后缀形式表达式的求值中,自然不可避免地会调用递归函数:首先在外部指定所有的操作,然后在表达式中清楚地说明构成部分。另一方面,在后缀式中首先出现操作数,而且整个表达式以“自底向上”的方式从简单的操作数和内部运算符逐渐地构造出来,因此,使用栈的迭代程序对后缀式似乎更加自然。(当然为任一种形式编写递归或非递归的程序是可能的,我们在这里只讨论动机,或者首先哪种方式看起来更加自然。)

* * * 栈

为了求后缀形式表达式的值,有必要记住操作数,直至后来某个时刻最终发现它们的运算符。记住它们的自然方法是将它们放入栈中,然后当遇到第一个运用的运算符时,这个运算符会在栈顶找到自己的操作数,如果它将结果仍放回栈中,则它的结果会处于后面那个运算符的一个操作数的正确位置上。当完成求值时,栈中仅存的值将是最终结果。这样我们得到求后缀表达式的一个函数。

此时我们要指出后缀和前缀表达式之间的一个重要区别:在前缀函数中,没有必要显式地检查是否已到达表达式的结尾,因为整个表达式自动地构成了第一个运算符的操作数;而从左至右读取后缀表达式时,我们会遇到子表达式,它们自己就是合法的后缀表达式。例如,如果我们在下面的表达式中‘↑’处停止读取,我们发现它已经是一个合法的后缀表达式。

$$b\ 2\ \uparrow\ 4\ a\ \times\ c\ \times\ -$$

为弥补这一问题，我们假设用某个特殊标记如';'标识一个后缀表达式的结束。当然，这个新的标记不属于 Token_type 所允许的先前种类中的任何一种。因此，我们要增加枚举的 Token_type 类型，使它包含一个新值 end_expression。其他的 C++类和辅助函数与前缀求值方法中所用的一样。

```
typedef Value Stack_entry;  //  Set the type of entry to use in stacks.
Error_code Expression::evaluate_postfix(Value &result)
/* Post: The tokens in Expression up to the first end_expression symbol are removed. If
        these tokens do not represent a legal postfix expression, a code of fail is
        returned. Otherwise a code of success is returned, and the removed sequence of
        tokens is evaluated to give Value result. */
{
  Token t;                    //  Current operator or operand
  Stack operands;             //  Holds values until operators are seen
  Value the_argument, first_argument, second_argument;
  do {
    if(get_token(t)==fail) return fail; //  No end expression token
    switch(t.kind()) {
    case unaryop:
      if(operands.empty()) return fail;
      operands.top(the_argument);
      operands.pop();
      operands.push(do_unary(t, the_argument));
      break;
    case binaryop:
      if(operands.empty()) return fail;
      operands.top(second_argument);
      operands.pop();
      if(operands.empty()) return fail;
      operands.top(first_argument);
      operands.pop();
      operands.push(do_binary(t, first_argument, second_argument));
      break;
    cose operand:
      operands.push(get_value(t))
      break;
    case end_expression:
      break;
    }
  } while(t.kind()!=end_expression);
  if(operands.empty()) return fail;
  operands.top(result);
  operands.pop();
  if(!operands.empty()) return fail;//  surplus operands detected
  return success;
}
```

13.3.5 程序的证明：栈元素计数

迄今为止，我们只给出了上述程序的非形式化的动机，并不清楚它在每种情况中是否都

能产生正确的结果。幸运的是，给出此程序形式化的证明且同时发现一个判断表达式是否用后缀式正确书写的有用标准并不困难。

我们使用的方法是跟踪栈中元素的数目。当得到每个操作数时，立即将它入栈。一元运算符首先执行出栈，然后再将结果入栈，因此并不改变栈中元素的数目。二元运算符做两次操作数出栈，再将结果执行一次入栈，因此栈中净减一个元素。更正式地，我们有下面的结论：

* * * 运行和条件

对由操作数、一元运算符和二元运算符构成的序列 E，由此形成运行和：在 E 的左端从 0 开始，对每个操作数视为 +1，每个一元运算符视为 0，每个二元运算符视为 -1。如果这个运行和从不小于 1，并且在 E 的右端正好是 1，则 E 满足运行和条件。

图 13.2 中说明了一个表达式的后缀式的运行和序列。

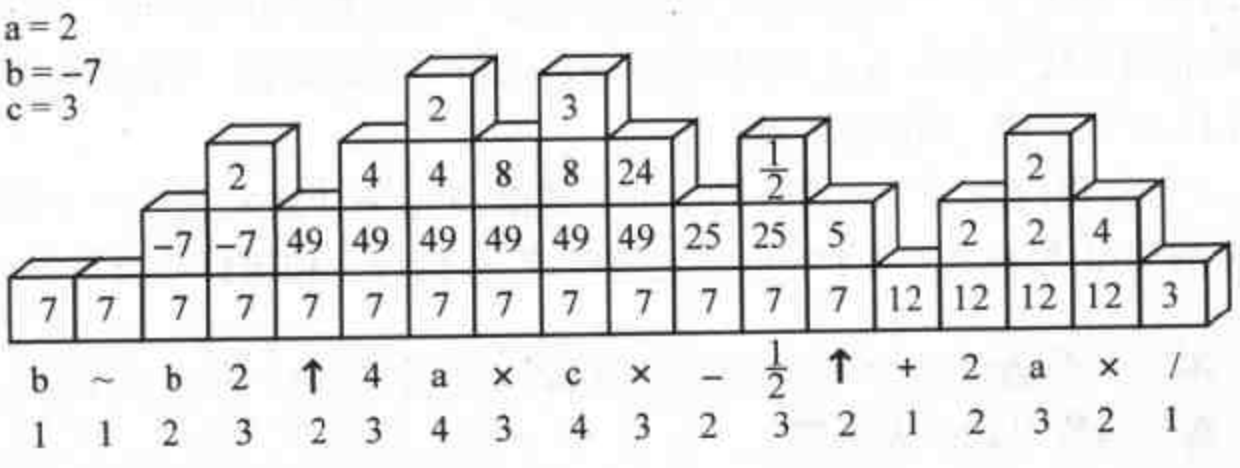


图 13.2　二次公式的栈帧和运行和

我们将同时证明下面两个定理。

定理 13.1　如果 E 是正确构成的后缀形式的表达式，则 E 必须满足运行和条件。

定理 13.2　正确构成的后缀形式的表达式可以用 evaluate_postfix 方法正确地求值。

证明　我们将通过对待求值的表达式 E 的长度进行数学归纳来同时证明这两个定理。

* * * 归纳证明

这个归纳的起始点是 E 仅属于单个操作数且长度为 1 的情况，此时这个操作数提供 +1，因此满足运行和条件。当对单个的操作数应用 evaluate_postfix 方法时，我们得到它的值并将它入栈(栈在此之前是空的)，在最后一次出栈时，它是这个方法的最终值，因此，对它的求值是正确的。

* * * 归纳假设

对归纳假设，我们现在假设 E 是长度大于 1 的正确构成的后缀表达式，程序对所有长度小于 E 的后缀表达式都能正确地求值，所有这些较短的表达式都满足运行和条件。由于 E 的长度大于 1，则 E 在最后一步的构造是 $F\ op$ 或 $F\ G\ op$。如果是 $F\ op$，则其中 op 是一元运算符且 F 是后缀表达式；如果是 $F\ G\ op$，则 op 是二元运算符且 F 和 G 是后缀表达式。

在这两种情况中，F 和 G 的长度都小于 E 的长度，因此由归纳假设，F 和 G 都满足运行

和条件，且 evaluate_postfix 方法能分别求出它们的值并获得正确的结果。

＊＊＊一元运算符

首先，考虑 *op* 是一元运算符的情况。由于 F 满足运行和条件，则结束时这个和正好是+1。*op* 作为一元运算符，对这个和提供 0，因此整个表达式 E 满足运行和条件。类似地，当这个方法到达 F 的结尾时，由归纳假设，它将正确地求出 F 的值并将它的值作为惟一的栈元素，之后一元运算符 *op* 应用到这个值上，它作为最终结果出栈。

＊＊＊二元运算符

最后，考虑 *op* 是二元运算符的情况，即 E 具有 $F\ G\ op$ 这样的形式，其中 F 和 G 是后缀表达式。当此方法到达 F 的最后一个标记时，则 F 的值将是栈中惟一的元素。类似地，它的运行和是 1。在下一个标记处，程序开始求 G 的值。由归纳假设，G 的求值也将是正确的，并且它的单独的运行和不会小于 1，且恰以 1 结束。由于 F 结束时运行和是 1，因此联合的运行和不会小于 2 且在 G 的结尾处恰好以 2 结束。因此 G 的求值将继续，不会影响栈底作为 F 结果的惟一的元素。当此表达式到达最终的二元运算符 *op* 时，运行和正好从 2 减为 1，这个运算符准确从栈中找出自己的两个操作数，求值后它给出惟一的结果，这样就完成了定理 13.1 和 13.2 的证明。（**证明结束**）

定理 13.1 允许我们通过对运行时栈中的元素数目保持计数来验证一个标记序列实际上是正确构成的后缀表达式。由于它的逆定理也成立，因此这种错误检查特别有用：

定理 13.3 如果 E 是满足运行和条件的操作数和运算符的任意序列，则 E 是一个正确构成的后缀形式的表达式。

证明 我们将再次使用数学归纳法证明定理 13.3，它的起始点是仅含一个标记的表达式。由于长度为 1 的序列的运行和（以及最终的和）是 1，因此这个标记必定是一个简单的操作数，而简单的操作数本身实际上是一个在语句构成上正确的表达式。

＊＊＊归纳证明

＊＊＊情况一：操作数

对于归纳步骤，现在假设对所有严格比 E 短的表达式已经验证此定理成立，E 的长度大于 1。如果 E 的最后一个标记是操作数，则它对运行和提供+1，并且由于最终的和为 1，因此结束之前的一个步骤这个和为 0。这与我们关于运行和条件满足的假设相反，因此 E 的最后一个标记必定是运算符。

＊＊＊情况二：一元运算符

如果此运算符是一元运算符，则它能被忽略，并且剩余的序列仍满足运行和条件。因此，由归纳假设，它是一个在语句构成上正确的表达式，且所有的 E 也是在语句构成上正确的表达式。

＊＊＊情况三：二元运算符

现在假设最后一个标记是二元运算符 *op*。为证明 E 是语句构成上正确的，我们必须使用运行和条件找出 *op* 的第一个操作数在序列中终止的位置和第二个操作数在序列中开始的位置。由于运算符 *op* 对这个和提供 -1，则在结束前一个步骤它是 2。这个 2 意味着栈中有两项，即 *op* 的第一和第二个操作数。当我们在序列 E 上向后回退时，最终到达栈中仅剩

一个元素时的某个位置(运行和是 1)，并且这个元素就是 op 的第一个操作数，因此打断此序列的位置是在运行和正好为 1 之处的尾部之前的最后一个位置上。由于我们在 E 的左部远端(如果不是在 E 之前的话)会找到为 1 的运行和，所以这个位置必定存在。当我们在最近的运行和为 1 的位置打断 E 的时候，E 就具有 $F\ G\ op$ 的形式。序列 F 满足运行和条件，并且以运行和 1 结束，因此由归纳假设，F 是正确构成的后缀表达式。由于在 $F\ G\ op$ 中的 G 期间运行和决不会再次变成 1，且恰在 op 前以 2 结束，我们可以从它们当中的每一个都减去 1 并推断 G 的单独的运行和满足条件。因此由归纳假设，G 也是一个正确构成的后缀表达式。由此，F 和 G 都是正确的表达式，并且可以由二元运算符 op 组合成正确的表达式 E，因此完成这个定理的证明。(**证明结束**)

我们可以对这个定理更进一步，证明运行和 1 出现的最近位置是能够将 E 分成在语句构成上正确的表达式 F 和 G 的惟一位置。我们假设它可以在任何位置分割。如果在 F 的结尾运行和不是 1，则 F 不是在语句构成上正确的表达式。如果运行和为 1 是在 F 的结尾，但是在 $F\ G\ op$ 的 G 部分运行期间再次达到 1，则在那一个位置 G 单独的运行和是 0，故 G 不是在语句构成上正确的表达式。现在已证明只有一种方式再现二元运算符的两个操作数，显然只有一种方式再现一元运算符的单个操作数。因此可以从表达式的后缀式再现它的中缀式以及这些操作完成的次序，我们通过使用另一对括号括起中缀式中每个操作的结果去表示它。

因此我们证明了下面的定理：

定理 13.4　满足运行和条件的后缀形式的表达式恰好对应一个完全用括号括起的中缀式的表达式。因而没有必要使用括号获得后缀形式表达式的惟一表示。

类似的定理对前缀表达式也成立，它们的证明留作练习。前面的定理同时提供了使用波兰表示法的理论证明和检查表达式语法正确的一种方便的办法。

13.3.6　后缀表达式的递归求值

许多人发现用于前缀表达式求值的递归函数比用于后缀表达式求值的基于栈的非递归函数更易于理解。在这一(可选的)节我们将证明为有利于后缀求值的递归而如何去消除栈。

然而，我们了解一下为何自然的方法会导致前缀求值而不是后缀求值的递归函数。可以用图 13.3 中的语法图描述前缀和后缀表达式，在这两种情况中，存在三种可能：表达式仅由单个操作数构成，或者表达式最外层的运算符是一元的，或者表达式是二元的。

* * * 前缀求值

我们在从头往后追踪图中的前缀式时，遇到的第一个表达式标记将决定应采取 3 个分支当中的哪一个，然后再没有更多的选择(除非在递归调用内部，这种情况现在不予考虑。)因此前缀求值的递归函数的结构与语法图非常近似。

* * * 后缀求值

然而在后缀图中，没有办法从第一个标记(它通常是一个操作数)区分要采取 3 个分支

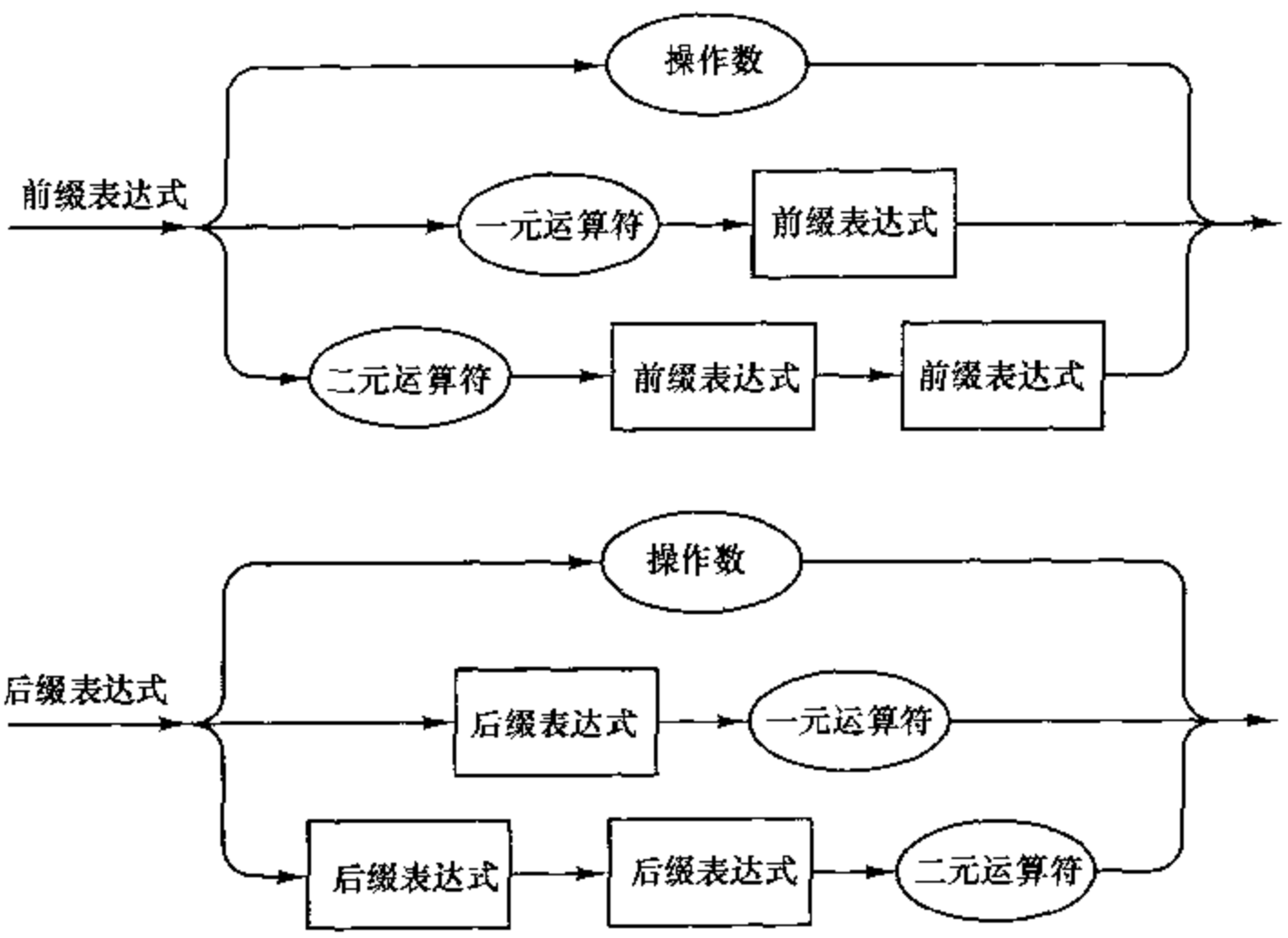


图 13.3　波兰表达式的语法图

中的哪一个，只有在遇到最后一个标记时才能确定用哪个分支。但这一事实也带来一种简单的递归解决方案：从右至左读取表达式，将语法图中的所有箭头反转过来，并使用本质上与前缀求值相同的函数！（当然，在新的函数中，我们必须反转运算符如－和/的参数的次序。）

但是，如果愿意的话，也可以按从左至右的自然方式读取表达式，但接下来就需要更多的工作。我们分别考虑后缀式中的 3 种标记。我们已知道表达式中的第一个标记必须是操作数，这一结论直接来自于第一个标记后的运行和（至少）是 1 的事实。由于一元运算符不改变运行和，一元运算符可以插入到初始操作数后面的任何位置。正是对作为第三种情况的二元运算符的研究产生了解决方案。

＊＊＊运行和

考虑运行和序列及序列中的和从 2 降为 1 的位置。由于二元运算符对和提供 -1，因此如果后缀表达式包含二元运算符，则必定存在这样的位置，并且它们一定对应表达式中二元运算符的两个操作数构成左边的完整表达式的位置。在图 13.2 的栈帧中说明了这样的情况。栈底的元素是第一个操作数，高度至少为 2 的位置序列，正好从高度 2 开始并在 2 结束，构成了第二个操作数的计算。孤立地看，这个序列本身正好形成后缀表达式。高度从 2 降到 1 标志着我们感兴趣的一个二元运算符。

在二元运算符后，可能出现更多的一元运算符，于是这个过程在跟有二元和一元运算符的自包含的后缀表达式的更多序列上重复其自身（如果运行和再次增长）。总而言之，我们已说明后缀表达式可以由图 13.4 的语法图描述，可以轻易地将它转换成下面的递归函数，其中的 C++约定与前面函数中的约定一样。

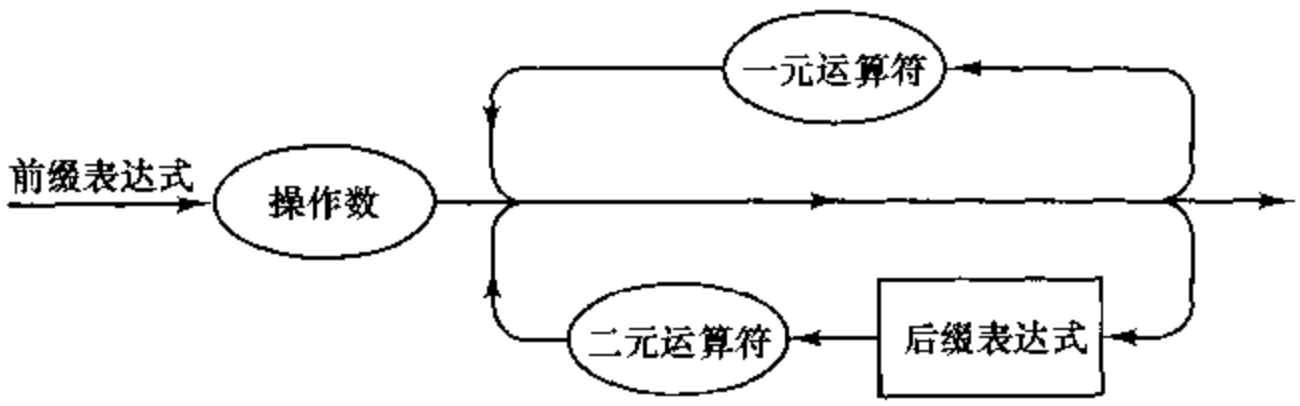


图 13.4　后缀表达式可供选择的语法图

*** 左递归**

图 13.3 的后缀图中出现的情况称为左递归，在转换到图 13.4 中的图表时，我们所采取的步骤与消除左递归所需的那些步骤是一样的。

首先是一个初始化递归的函数。

```
Error_code Expression::evaluate_postfix(Value &result)
/*Post: The tokens in Expression up to the first end_expression symbol are removed. If
        these tokens do not represent a legal postfix expression, a code of fail is
        returnod. Otherwise a code of success is returned, and the removed sequence of
        tokens is evaluated to give Value result.*/
{
  Token first_token,final_token;
  Error_code outcome;
  if(get_token(first_token)==fail||first_token.kind()!=operand)
     outcome=fail;
  else {
    outcome=recursive evaluate(first_token,result,final_token);
  if(outcome==success && final_token.kind()!=end_expression)
      outcome=fail;
  }
  return outcome;
}
```

真正的递归使用从表达式剩余部分分离出来的第一个标记的值。

```
Error_code Expression::recursive_ evaluate(const Token &first_token,
                                           Value &result,Token &final_ token)
/*Pre: Token first_token is an operand.
  Post: If the first_token can be combined with initial tokens of the Expression to
        yield a legal postfix expression followed by either an end _expression symbol or
        a binary operator, a code or success is returned, the legal postfix subexpression
        is evaluated, recorded in result, and the terminating Token is recorded as final
        _token. Otherwise a code of fail is returned. The initial tokens of Expression
        are removed.
  Uses: Methods of classes Token and Expression, including recursive_evaluate and
        functions do_unary, do_binary, and get_value.*/
{
  Value first_segment=get_value(first_token),
        next_segment;
  Error_code outcome;
  Token current_token;
```

```
Token_type current_type;
do {
  outcome=get_token(current_token);
  if(outcome!=fail){
    switch(current_type=current_token.kind()){
    case binaryop:          //  Binary operations terminate subexpressions.
    case end_expression:    //  Treat subexpression terminators together.
      result=first_segment;
      final_token=current_token;
      break;
    case unaryop:
      first_segment=do_unary(current_token,first_segment);
      break;
    case operand:
      outcome=recursive_evaluate(current_token,
                                 next_segment,final_token);
      if(outcome==success && final_token.kind()!=binaryop)
         outcome=fail;
      else
        first_segment=do_binary(final_token,first_segment,
                                next_segment);
      break;
    }
  }
} while(outcome==success && current_type!=end_expression &&
                            current_type!=binaryop);
return outcome;
}
```

练习 13.3

E1. 以(1)非递归和(2)递归版本的 evaluate_postfix 函数跟踪下列每个表达式的操作。对递归函数，画出递归调用树，在每个结点处指示是在处理哪个标记。对非递归函数，画出栈帧序列，表明在每个阶段处理哪个标记。

(1)$a\ b+c\ \times$

(2)$a\ b\ c+\times$

(3)$a!\ \ b!\ /c\ d\ -a!-\times$

(4)$a\ b<!\ \ c\ d\ \times<e\ \|$

E2. 通过画出递归调用树表明在每个阶段处理哪个标记，跟踪 evaluate_prefix 函数在下列每个表达式上的操作。

(1)$/+x\ y!\ n$

(2)$/+!\ x\ y\ n$

(3)$\&\&<x\ y\ \|\ !=+x\ y\ z><x\ 0$

E3. 下面的哪个表达式是在语句构成上正确的后缀表达式？指出每个不正确的表达式中的错误。将每个正确的表达式翻译成中缀式，必要时使用圆括号去避免多义性。

(1)$a\ b\ c+\times\ a/c\ b+d/-$

(2)$a\ b+c\ a\ \times\ b\ c/d-$

(3)*a b+c a* ×-c ×+*b c*-

(4)*a*~*b* ×

(5)*a* × *b* ~

(6)*a b* × ~

(7)*a b* ~ ×

E4. 将下列每个表达式从前缀式转换成后缀式。

(1)/+*x y*! *n*

(2)/+!*x y n*

(3)&&<x y ‖ !=+x y z><x 0

*E*5. 将下列每个表达式从后缀式转换成前缀式。

(1)a b+c ×

(2)*a b c*+×

(3)*a*!*b*!/*c d* -*a*!-×

(4)*a b*<!*c d* ×<*e* ‖

E6. 对前缀式的表达式，系统地阐述并证明与定理(1)13.1；(2)13.3；(3)13.4 类似的定理。

13.4　从中缀式到波兰形式的转换

很少有程序设计员会习惯采用波兰形式书写代数或逻辑表达式，虽然这样做会比通常的前缀式更加一致和富有逻辑。为方便使用我们为波兰表达式求值而设计的算法，我们还必须设计另一个有效的方法将中缀式的任意表达式转变成波兰表示。

作为最初的简化，我们只考虑将中缀表达式转换成后缀式的算法；其次我们不考虑放在自己操作数右侧的一元运算符，在设计算法时这些运算符不会造成概念上的困难，但它们会使编写出来的函数稍微复杂一些。

为设计这个算法，可能考虑的一种方法是：首先从中缀式建立表达式树，然后遍历这棵树以获得后缀式。然而，从中缀式建立表达式树实际上比直接构造后缀式更加复杂。

在后缀式中，因为所有的运算符都在自己的操作数之后，则从中缀式到后缀式转换的任务等于移动运算符使它们位于其操作数之后而不是之前或之间。换句话说：

* * * 滞后运算符

滞后每个运算符，直到其右侧的操作数已被转换。每个简单的操作数无滞后地输出。

图 13.5　中说明了这个操作。

我们必须解决的主要问题是什么标记将终止给定运算符右侧的操作数，并从而标识出运算符应放置的位置。为完成这个任务，必须同时考虑括号和运算符的优先级。

第一个问题比较简单。如果在操作数中存在一个左括号，则直到(并包括)匹配右括号的每个字符也必须包含在此操作数中。对于第二个考虑运算符优先级的问题，我们要独立于优先级为 6 的运算符即一元和求幂运算符去考虑二元运算符，其原因在于优先级为 6 的运算符是自右向左进行求值，而较低优先级的二元运算符是自左向右求值。

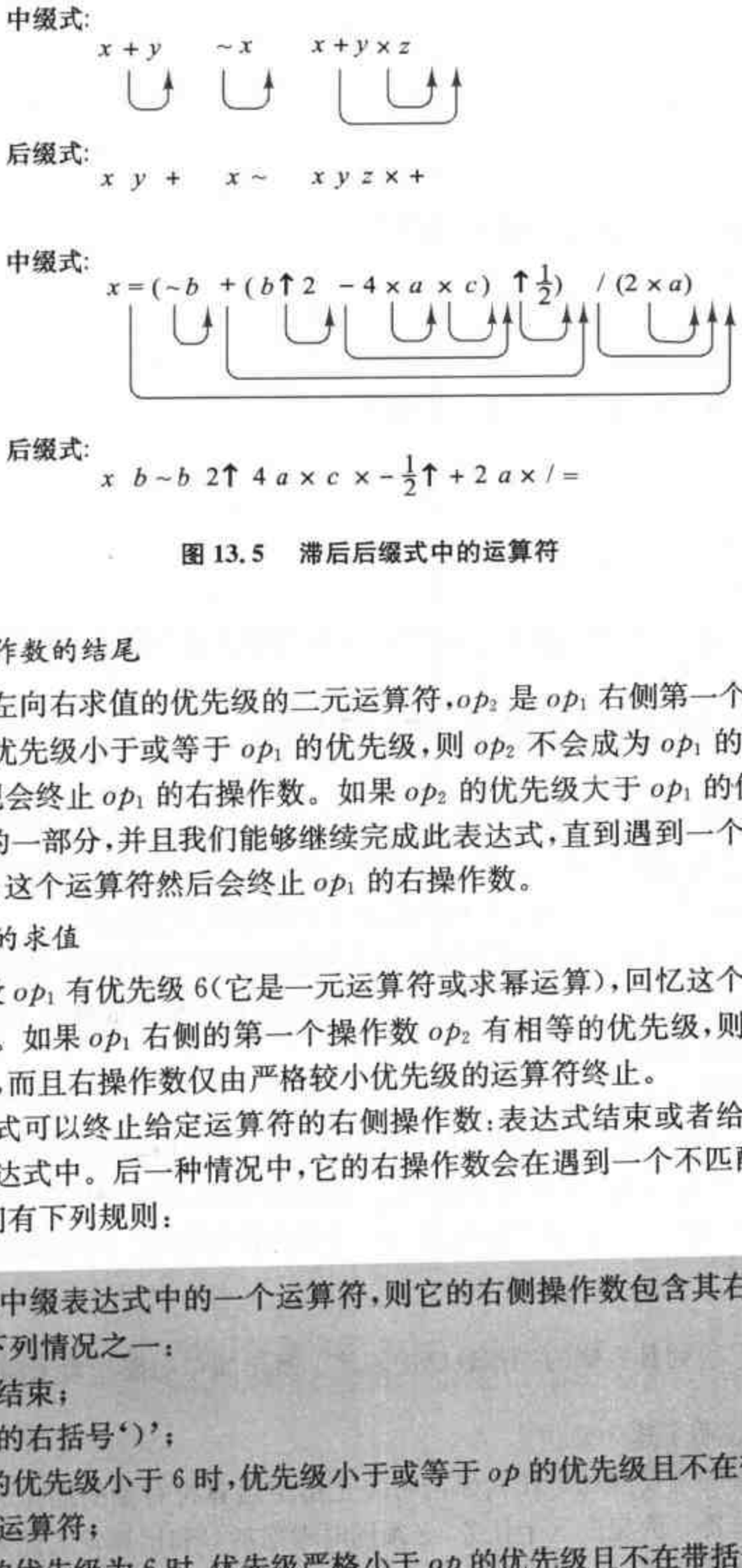


图 13.5 滞后后缀式中的运算符

* * *找出右操作数的结尾

设 op_1 是自左向右求值的优先级的二元运算符，op_2 是 op_1 右侧第一个不带括号的运算符。如果 op_2 的优先级小于或等于 op_1 的优先级，则 op_2 不会成为 op_1 的右操作数的一部分，并且它的出现会终止 op_1 的右操作数。如果 op_2 的优先级大于 op_1 的优先级，则 op_2 是 op_1 的右操作数的一部分，并且我们能够继续完成此表达式，直到遇到一个优先级小于或等于 op_1 的运算符，这个运算符然后会终止 op_1 的右操作数。

* * *自右向左的求值

接下来，假设 op_1 有优先级 6(它是一元运算符或求幂运算)，回忆这个优先级的运算符是自右向左求值。如果 op_1 右侧的第一个操作数 op_2 有相等的优先级，则它将是 op_1 的右操作数的一部分，而且右操作数仅由严格较小优先级的运算符终止。

还有两种方式可以终止给定运算符的右侧操作数：表达式结束或者给定运算符本身是在带括号的子表达式中。后一种情况中，它的右操作数会在遇到一个不匹配的右括号‘)’时结束。总之，我们有下列规则：

如果 *op* 是中缀表达式中的一个运算符，则它的右侧操作数包含其右部的所有标记，直到遇到下列情况之一：

(1)表达式结束；

(2)不匹配的右括号‘)’；

(3)当 *op* 的优先级小于 6 时，优先级小于或等于 *op* 的优先级且不在带括号的子表达式中的运算符；

(4)当 *op* 的优先级为 6 时，优先级严格小于 *op* 的优先级且不在带括号的子表达式中的运算符。

＊＊＊运算符栈

从这些规则，我们能发现记住被滞后的运算符的合适办法是将它们保存在栈中。如果运算符 op_2 出现在运算符 op_1 的右侧，但优先级较高，则 op_2 将在 op_1 之前输出。因而这些运算符是以后进先出的次序输出。

为这种转换编写一个算法的关键是稍微改变我们的观点：当每个标记出现在输入中时，询问一下先前被滞后的运算符(即在栈中)中的哪些现在因新的标记而终止了自己的右操作数，以至于现在已经到了将它们移入输出中的时候。前面的条件于是变成下面这些条件：

＊＊＊出栈

(1)在表达式的结尾，所有的运算符被输出。
(2)一个右括号导致自其相应左括号开始出现的所有运算符被输出。
(3)优先级不是 6 的运算符导致所有更大或相等优先级的其他运算符被输出。
(4)优先级为 6 的运算符不导致运算符被输出。

为实现第二条规则，我们要将遇到的每个左括号放进栈中，然后当匹配的右括号出现时，从栈中弹出这些运算符，这一对括号可以被丢弃。

我们现在可以将这些规则并入函数中。为此，除了现在的方法 Token_type Token::kind()能返回如下两个另外的结果之外，我们需要使用上一节中同样的辅助类型和函数：

leftparen　　rightparen

它们分别表示左括号和右括号。现在的栈包含的是标记(运算符)而不是值。

除了得到来自输入(中缀表达式)的下一个标记的方法 Expression::get_token()外，我们使用另一个方法 **void** Expression::put_token(**const** Token &t)，它将给定的标记放到(后缀)表达式的尾部。因而依赖于期望的应用，这两个方法可以对文件读写，或者也可能仅引用已经创建的表。

最后，我们要用到一个新函数 priority(**const** Token &operation)，它将返回代表运算符的任何 Token 的优先级。

由这些约定，我们能编写出将表达式从中缀式转换成后缀式的方法。在这一实现中，我们已避免了检查原始表达式是否合法这个问题，为此，我们必须将这个假设作为此方法的前置条件加入。

```
Expression Expression::infix_to_postfix()
/*Pre: The Expression stores a valid infix expression.
  Post: A postfix expression that translates the infix expression is returned.*/
{
  Expression answer;
  Token current,prior;
  Stack delayed_operations;
  while(get_token(current)!=fail){
    switch(current.kind()){
    case operand:
      answer.put_token(current);
      break;
    case leftparen:
```

```
    delayed_operations.push(current);
    break;
  case rightparen:
    delayed_operations.top(prior);
    while(prior.kind()!=leftparen){
      answer.put_token(prior);
      delayed_operations.pop();
      delayed_operations.top(prior);
    }
    delayed_operations.pop();
    break;
  case unaryop:
  case binaryop:          //  Treat all operators together.
    bool end_right=false; //  End of right operand reached?
    do {
      if(delayed_operations.empty())end_right=true;
      else {
        delayed_operations.top(prior);
        if(prior.kind()==leftparen)end_right=true;
      else if(prior.priority()<current.priority())end_right=true;
      else if(current.priority()==6)end_right=true;
      else answer.put_token(prior);
      if(!end_right)delayed_operations.pop();
      }
    } while(!end_right);
    delayed_operations.push(current);
    break;
  }
  while(!delayed_operations.empty()){
    delayed_operations.top(prior);
    answer.put_token(prior);
    delayed_operations.pop();
  }
  answer.put_token(";");
  return answer;
}
```

* * * 例子

作为这个算法的例子，图 13.6 显示了将二次公式 $x=(\sim b+(b^2-4\times a\times c)^{\frac{1}{2}})/(2\times a)$转换成后缀式执行的步骤。(记住我们使用'~'表示一元取反。)

这就完成了转换成后缀式的讨论。显然转换成前缀式的描述会具有明显的相似性，但由于欧式语言中从左到右读取这一表面上不相关的事实而引起一些困难。如果我们打算从左至右将一个表达式转换成前缀式，则不仅需要重排运算符，而且需要滞后操作数直到它们的运算符被输出之后。但是操作数的相对次序在转换中不会改变，因此保存操作数的合适的数据结构不是栈(它实际上是队列)。既然栈不完成这个工作，具有非显式辅助存储的递归程序也就不会完成，因为这两种程序能完成相同的任务。因而从左向右转换到前缀式需要不同的方法。诀窍是使用与我们已设计的从左向右的后缀转换非常类似的方法，通过从右向左处理表达式而转换成前缀式。这些细节都留作练习。

输入标记	栈内容 (最右侧标记位于栈顶)	输出标记
x		x
=	=	
(	=(	
~	=(~	
b	=(~	b
+	=(+	~
(	=(+(	
b	=(+(	b
↑	=(+(↑	
2	=(+(↑	2
-	=(+(-	↑
4	=(+(-	4
×	=(+(-×	
a	=(+(-×	a
×	=(+(-×	×
c	=(+(-×	c
)	=(+	× -
↑	=(+↑	
$\frac{1}{2}$	=(+↑	$\frac{1}{2}$
)	=	↑ +
/	=/	
(	=/(	
2	=/(	2
×	=/(×	
a	=/(×	a
)	=/	×
表达式结束		/=

图 13.6　将二次公式转换成后缀式

练习 13.4

E1. 设计一个将表达式从前缀式转换成后缀式的方法，使用这一章中的 C++约定。

E2. 编写一个将表达式从后缀式转换成前缀式的方法，使用这一章中的 C++约定。

E3. 完全带括号的表达式具有下列形式之一：

(1)一个简单的操作数；

(2)($op\ E$)，其中 op 是一元运算符且 E 是完全带括号的表达式；

(3)($E\ op\ F$)，其中 op 是二元运算符且 E 和 F 都是完全带括号的表达式。

因此，在完全带括号的表达式中，每个操作的结果被括在括号中。完全带括号的表达式的例子如$((a+b)-c)$，$(-a)$，$(a+b)$，$(a+(b+c))$。编写一个方法将表达式从(1)前缀式和(2)后缀式转换成完全带括号的形式。

E4. 重新将 infix_to_postfix 编写成一个不使用栈或其他数组的递归函数。

程序设计项目 13.4

P1. 为波兰表达式构造一个菜单驱动的演示程序。此程序的输入是任意中缀、前缀或后缀形式的表达式。程序然后将按用户请求将表达式转换成任意完全带括号的中缀、前缀或后缀形式并输出结果。操作数应当仅是单个字母或数字，允许的运算符包括：

二元运算符：+ - * / ∧ : < > & | =

左一元运算符：# ~

右一元运算符：! ′ ″

使用 13.1.1 的表格中给出的优先级而不是 C++的优先级。此外，此程序仅在中缀表达式中允许使用括号‘(’和‘)’。在本项目中使用的一些特殊符号是：

& 布尔与(同 C++中的 &&)　　| 布尔或(同 C++中的 ||)

: 赋值(同 C++中=)　　% 模(二元运算符)

∧ 幂(同↑)　　! 阶乘(在右部)

′ 导数(在右部)　　″ 二次导数(在右部)

~ 一元取反　　# 布尔非(同 C++中的!)

13.5 一个交互式的表达式求值程序

计算程序运行期间交互键入的函数的值的程序有许多应用。其中的一个应用是画数学函数图的程序。假设你正编写这样的一个程序，用于帮助一年级学微积分的学生画出函数图。这些学生中的大多数不知道如何编写或编译程序，因此你希望在自己的程序中包含某种办法，使得程序运行时，用户能为函数提交这样的表达式：$x * \log(x) - x \wedge 1.25$。然后，程序能根据适宜的 x 值画出它的图形。

* * * 目标

这一节的目标是描述这样的一个程序。我们特别关注有助于解决此问题的两个子程序，其中第一个子程序接受从终端键入作为输入的表达式，表达式包含有常量、变量、算术运算符以及标准函数，允许使用括号。然后将这个表达式转换成后缀式并将它保存在一个标记表中。第二个子程序将使用作为调用参数给出的变量值对这个后缀表达式求值并返回结果，然后这个结果可用于画图。

* * * 目的

* * * 健壮性

我们着手进行这个项目具有几个原因。它表明如何采用为处置波兰表示法而设计的思想并将这些思想创建成完整、具体和能运行的程序。这样，本项目说明了程序设计的一种问题求解方法，在这种方法中，我们从关键问题的解决方案入手并完成所需的辅助函数的结构。最后，由于这个项目打算供给几乎没有计算机经验的人使用，因此它提供了测试健壮性的机会，即程序经受意外或错误输入而不导致灾难性失败的能力。

13.5.1　总体结构

为给用户提供修改画图的灵活性，我们使用菜单驱动的程序，因此主程序的操作具有下列常见形式：

```
int main()
/* Pre:  None
   Post: Acts as a menu-driven graphing program.
   Uses: Classes Expression and Plot, and functions introduction, get_command, and do_
         command.*/
{
   introduction();
   Expression infix;                    //  Infix expression from user
   Expression postfix;                  //  Postfix translation
   Plot graph;
   char ch;
   while((ch=get command())!='q')
     do_command(ch,infix,postfix,graph);
}
```

在主程序中，我们使用一对 Expression 对象存放用户的中缀表达式及其后缀转换。我们也使用 Plot 类控制所有的画图活动。在各种方法和函数间进行工作划分是 do_command 的任务：

```
void do_command(char c,Expression &infix,Expression &postfix,Plot &graph)
/* Pre: None
   Post: Performs the user command represented by char c on the Expression infix, the
         Expression postfix, and the Plot graph.
   Uses: Classes Token, Expression and Plot.*/
{
   switch(c){
   case 'r':                            //  Read an infix expression from the user.
     infix.clear();
     infix.read();
     if(infix.valid_infix()==success)postfix=infix.infix_to_postfix();
     else cout<<" Warning:Bad expression ignored."<<endl;
      break;
   case'w':                             //  Write the current expression.
     infix.write();
     postfix.write();
     break;
   case 'g':                            //  Graph the current postfix expression.
     if(postfix.size()<=0)
        cout<<" Enter a valid expression before graphing! "<<endl;
     else {
       graph.clear();
       graph.find_points(postfix);
       graph.draw();
    }
    break;
    case 'l':                           //  Set the graph limits.
     if(graph.set_limits()!=success)
```

```
        cout<<" Warning:Invalid limits"<<endl;
      break;
    case 'p':                          //  Print the graph parameters.
      Token::print_parameters();
      break;
    case'n':                           //  Set new graph parameters.
      Token::set_parameters();
      break;
    case'h':                           //  Give help to user.
      help();
      break;
    }
}
```

作为对用户命令'r'的应答,这个程序请求输入一个表达式并将它分割成标记。valid_infix()方法判断这个表达式在语句构成上是否正确。如果正确,则用我们已学过的 infix_to_postfix()方法将它转换成后缀式。

作为对用户命令'g'的应答,为画出这个表达式的图形,我们用坐标 x 的多个不同的值求出后缀式的值,每个 x 值与上一个 x 值之间仅相差一个较小的增量 x_increment,然后在屏幕上依次将每个结果标绘成单个的点。

用户可以应用命令'l'选择要标绘图形的 x 的范围、使用的增量及显示表达式求值结果的范围。这些值都保存在 Plot graph 中,可以使用 Plot::set_limits()方法去重新设置。

* * * 表达式参数

除了用于标绘的(独立的)变量 x 以外,表达式中还可以包含更多的变量,我们称之为图的参数。例如表达式 a * cos(x)+b * sin(x)中,a 和 b 都是参数,这些参数在画某个图形时保持固定的值,但从一个图形到下一个图形时,可以用用户命令'n'去修改它们,而不用对表达式本身做出任何修改。这些参数将存为标记,因此我们提供一个静态的 Token 方法 set_parameters 去重新设置表达式中出现的所有参数的值。

我们的程序同时必须建立预定义标记(诸如运算符+,-,*,加上用于表达图形坐标的操作数 x,或许还有其他的常量)的定义。但是在我们能够确定这些初始化的细节之前,必须确定标记和表达式的数据结构。

13.5.2 数据的表达:类说明

我们关于数据结构的第一个决策是关心如何存储和检索波兰表达式中使用的标记。对每个不同的标记,必须记住:

- 它的名字(作为一个 String),以使我们能在输入表达式时识别它;
- 它的种类,为操作数、一元运算符、二元运算符和右一元运算符(象阶乘'!')、左括号或右括号之一;
- 对运算符而言的优先级;
- 对操作数而言的值。

将每个标记的表达看作是包含此信息的一个记录是合理的,但随之也带来一点小麻烦:同一个标记可能在一个表达式中出现几次。如果标记是一个操作数,则我们必须确保它每

次都给出同样的值。如果我们将记录本身放入表达式中，则将某个值赋给操作数时，必须确信与那个操作数对应的所有记录均得到更新。

＊＊＊词典

通过让每个标记与一个整数代码相关联，并将这个代码而不是整个记录放入表达式中，能够避免保持对给定变量的引用链。因此我们将为这些标记创建一个词典，它包含一个数组，这个数组采用这些整数代码作为自己的下标，而且将容纳这些标记的完整记录。这样，如果 k 是某个变量的代码，则此变量在表达式中的每次出现都将导致我们去查看词典中位置 k 的相应值，从而自动地保证每次都得到同样的值。

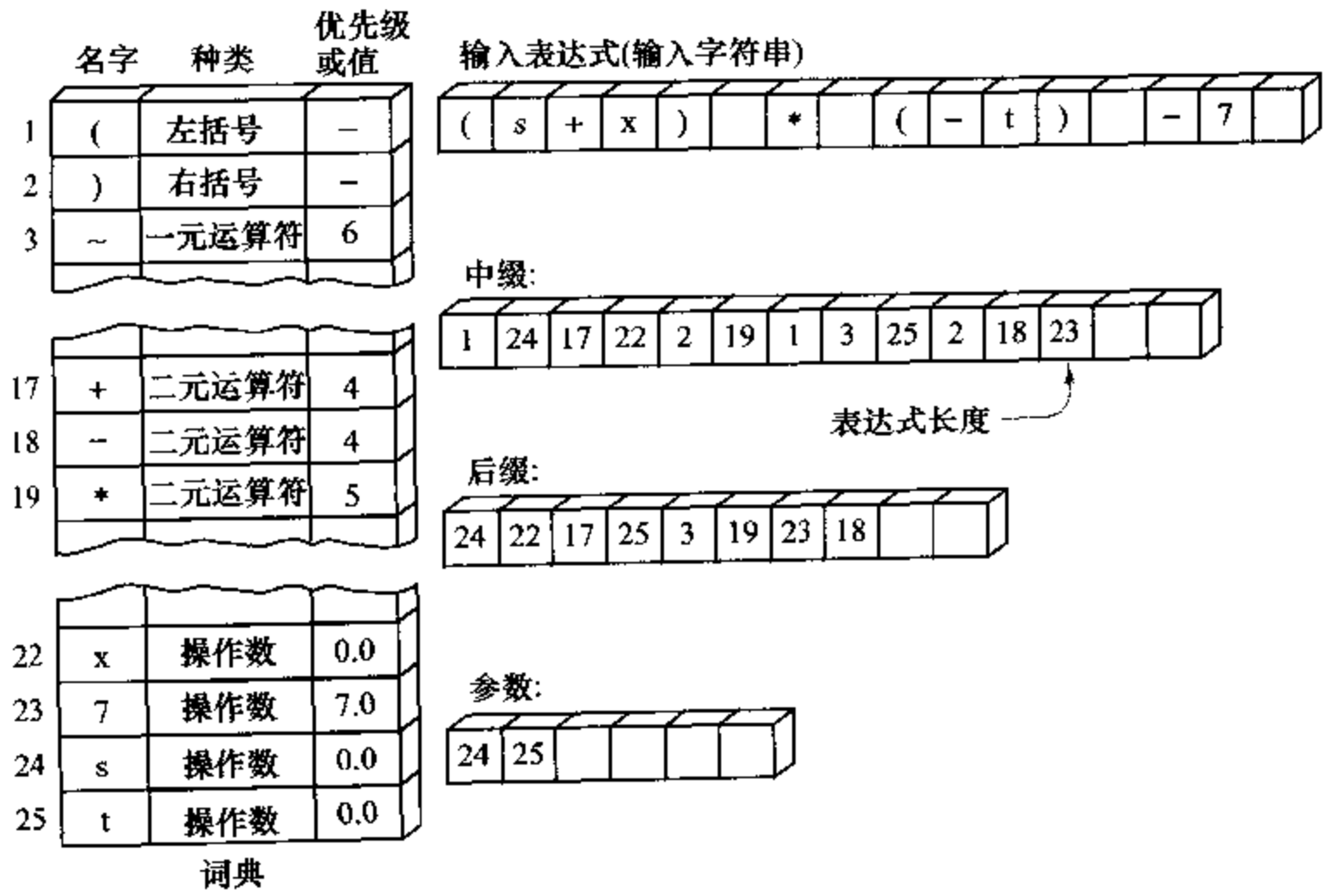


图 13.7　标记和表达式的数据结构

我们现在引入一种数据结构去容纳作为词典元素存储的标记记录：

```
struct Token_record {
  String name;
  double value;
  int priority;
  Token_type kind;
};
```

在这个结构中，用来标识一个标记的 name 成员采用 6.3 节中的 String 类实现，在那里，String 对象能被安全地复制、在赋值运算符间传递并使用普通的运算符进行比较。

＊＊＊哈希表

这个词典必须包含一个 Token_record 对象的数组，此外，它必须提供查找特定记录的方法，或者从它的整数代码进行查找，或者从它的名字进行查找。我们通过包含一个称为 index_code 的哈希表作为词典的成员，从而解决了这个信息检索问题。在此哈希表中，我们

只存储代码并使用这些代码去查看记录数组以查找具有给定名字的标记的全部信息。

在一个用户表达式中一般不会超过几十个标记，可以证明检索代码的最佳方式是顺序查找词典。顺序查找易于设计，不需要更多的数据结构，因此相对于更复杂的方法所需要的额外时间代价可以忽略不计。

* * * 符号表

然而，此项目的目的之一是举例说明较大的应用，其中顺序查找的时间代价不再能够被忽略不计。例如，在一种编译器中，可能必须识别几百种不同的符号，因而必须使用更复杂的符号表。一种较好的选择是使用哈希表将标记名映射成整数代码。

现在我们已得到下面的结构去表达词典：

```
struct Lexicon {
  Lexicon();
  int hash(const String &x) const;
  void set_standard_tokens();   //  Set up the predefined tokens.
  int count;                    //  Number of records in the Lexicon
  int index_code[hash_size];    //  Declare the hash table;
  Token_record token_data[hash_size];
};
```

* * * 静态数据成员

我们现在回到将要存储在表达式中的标记的表达。每个标记应包含一个整数代码，用以表达它的 Token_record 在词典 Lexicon 的 token_data 数组中的下标。这意味着每个标记 Token 需要访问 Lexicon，因此它能够检索其相关的记录。然而，我们意欲保护 Lexicon 及其记录不受其他的访问，从而确保它们的完整性。通过将 Lexicon 声明成 Token 类的静态数据成员，可以同时实现上述两个目标。静态数据成员仅被创建和存储一次，但它能作为这个类的任何对象的数据成员被访问。

我们现在设计出下面的 Token 类的概要：

```
class Token {
public:
//  Add methods here.
private:
   int code;
   static Lexicon symbol_table;
   static List<int> parameters;
};
List<int> Token::parameters; //  Allocate storage for static Token members.
Lexicon Token::symbol_table;
```

* * * parameters

这个类的概要中包含另一个静态数据成员 parameters，它保存表示参数的那些标记的整数代码表。当对一个输入表达式进行解码时，此程序会找出常量和新变量(参数)，它们随后将被加入到词典中，并且将被归为操作数，但如果是参数，则在表达式求值前会要求用户给出它们的值。为能够提示用户给出这些值，有必要保存与参数对应的那些标记的代码列表。由于我们需要从 Token 对象访问这个参数表，因此将 parameters 声明为 Token 类的另一个静态数据成员就显得很方便。

* * *静态成员的存储分配

这两个静态数据成员不占用任何 Token 对象中的存储空间,因此必须在类说明之外为它们分配存储空间。因而,我们在类说明之后用适当的定义为成员 parameters 和 symbol_table 保留存储空间。这些定义将在运行时、在 main 函数开始运行前处理。这保证我们在开始使用 parameters 对象和 symbol_table 对象之前,List 和 Lexicon 的构造函数已完成了所有要求的初始化操作。当然,当我们创建 Lexicon symbol_table 时,可能向 List parameters 中加入操作数,因此必须确信在声明 Lexicon symbol_table 之前声明 List parameters,这样就确保这个表在增加元素前已被合适地初始化。

向表达式中放入包含整数代码的标记而不是记录的做法具有节省空间的优点,但标记所用的空间未必是本项目设计的关键限制因素。用自变量 x 的多个不同的值去求后缀表达式的值所需要的时间更可能是昂贵的。

我们将继续使用早先作为标记表的表达式实现,这样就可以使用早先用来转换表达式和对表达式求值的高效方法。

```
class Expression {
public:
//   Add method prototypes.
private:
  List<Token>terms;
  int current_term;
//   Add auxiliary function prototypes.
};
```

图 13.7 中说明了所有这些数据结构以及在主要函数中使用的一些数据结构。与这些数据结构一道,还需要许多常量、辅助类型说明和类型标识,特别地,我们需要创建枚举类型 Token_type,声明和初始化 **const int** hash_size,并将类型标识符 Valuc 定义为 double 的同义词。同时我们还必须包含自己早先的 String 类、List 类和 Stack 类的实现,String 类和 List 类已出现在我们的标记和表达式的说明中。正如 13.4 节,在将表达式从中缀转换成后缀式时需要使用 Stack。最后,程序使用 Plot 类和其他辅助的画图结构,我们将在后面进行讨论和实现。我们现在考虑程序需要的类实现。

13.5.3　Token

* * * Token 方法

为与这一章早先设计的函数一致,我们将为 Token 类配备一个 kind()方法,用它报告 Token 的种类。为方便起见,我们将使用类似的 Token 的方法 priority(),name()和 value(),它们报告其他的标记信息。此外,code_number()方法返回一个标记的代码。

* * * Token 构造函数

我们经常需要从其标识 String 产生一个新的 Token 对象:换言之,我们需要重新将 String 改成 Token,我们能方便地将这一改造操作实现成一个带有 String 自变量的 Token 构造函数。我们也需要提供另一个不带自变量的 Token 构造函数,这个函数在我们声明但未初始化标记时被调用。最后,我们需要 Token 方法去设置和检查表示参数的标记的值,以及对坐标 x 赋值的方法:这些方法仅访问或修改静态数据成员并因此本身具有 static 型的

修改符。我们现在能确定下面的类：

```
class Token {
public:
   Token(){}
   Token(const String &x);
   Token_type kind()const;
   int priority()const;
   double value()const;
   String name()const;
   int code_number()const;
   static void set_parameters();
   static void print_parameters();
   static void set_x(double x_val);
private:
   int code;
   static Lexicon symbol _table;
   static List<int>parameters;
};
```

1. 访问 Token 信息

提供标记信息的方法只是查看 Lexicon，例如，kind()方法如下实现：

```
Token_type Token::kind()const
{
  return symbol_table.token_data[code].kind;
}
```

2. Token 构造函数和初始化

默认的不带参数的构造函数不对 Token 赋予初始的数据代码，因此用空的代码体实现。另一个构造函数带有一个 String 自变量，因此必须赋予一个合适的 Token 代码。这个 String 首先由 Lexicon:: hash 方法处理，此哈希操作包含冲突解决并返回数组 Lexicon:: index_code 中的一个位置 location。结果得到的元素 index_code[location]或者是早先赋给 String 的代码，或者是指示 String 未发现的特殊代码-1。在任何一种情况下，当数据位于新标记中时，构造函数都可以确定要使用的正确的初始化代码。

```
Token::Token(const String &identifier)
/* Post: A Token corresponding to String identifier is constructed. It shares its code
         with any other Token object with this identifier.
   Uses: The class Lexicon.*/
{
   int location=symbol_table.hash(identifier);
   if(symbol_table.index_code[location]==-1){ //  Create a new record.
      code=symbol_table.count++;
      symbol_table.index_code[location]=code;
      symbol_table.token_data[code]=attributes(identifier);
      if(is_parameter(symbol_table.token_data[code]))
         parameters.insert(0,code);
   }
   else code=symbol_table.index_code[location]; //  Code of an old record
}
```

* * * 函数 attributes

辅助函数 attributes 检查标记的名字并按我们的约定创建一个合适的 Token_record。例如，如果参数 String identifier 是“*”号，则函数 attributes 返回一个包含数据值 kind+ = binaryop 且 priority=5 的记录。attributes 的实现只包括一系列的赋值语句，我们将省略这些语句。我们要指出：函数 attributes 仅为每个标记提供静态的初始化，但词典为我们给出了关于标记的动态变化的信息。例如，attribute 总是将默认值 0.0 赋给一个参数，但词典将记录用户上次输入的任何值。

* * * 函数 is_parameter

另一个辅助函数 is_parameter 通过检查 Token 的标识符，确定 Token 是否是一个参数。

3. 参数值

* * * 函数 set_parameters

在对表达式求值前，我们需要为存在的参数创建值。实现此任务的函数 Token:: set_parameters 遍历参数表，打印关于每个元素的信息并从用户处请求更新的信息。

```
void Token :: set parameters ()
/* Post: All parameter values are printed for the user, and any changes specified by the
         user are made to these values.
   Uses: Classes List, Token_record, and String, and function read_num. */
{
   int n=parameters.size();
   int index_code;
   double x;
   for(int i=0; i<n; i++){
     parameters.retrieve(i,index_code);
     Token_record &r=symbol_table.token_data[index_code];
     cout <<" Give a new value for parameter "<<(r.name).c_str()
          <<" with value "<<r.value<<endl;
     cout<<" Enter a value or a new line to keep the old value:"<<flush;
     if(read_num(x)==success)r.value=x;
   }
}
```

13.5.4　词典

Lexicon 的构造函数必须将成员哈希表 index_code[]创建成空表，然后应用辅助方法 Lexicon:: set_standard_tokens 输入关于所有预定义标记的信息。因此构造函数采用如下形式：

```
Lexicon::Lexicon()
/* Post: The Lexicon is initialized with the standard tokens.
   Uses:set_standard_tokens*/
{
   count=0;
   for(int i=0; i<hash_size; i++)
```

```
    index_code[i]=-1;   //  code for an empty hash slot
  set_standard_tokens();
}
```

* * * 函数 set_standard_tokens

图 13.8 中显示了由 set_standard_tokens 输入的预定义标记的完整列表。注意，我们包含了并非计算机语言标准部分的操作(如以 2 为底的对数 lg)和诸如 e 和 π 这样的常量。我们在这一节感兴趣的表达式一般有实数结果，因此我们不包含任何布尔取值的操作。

标记	名字	种类	优先级/值	
0	;	表达式结束		
1	(	左括号		
2	)	右括号		
3	~	一元运算符	6	取反
4	abs	一元运算符	6	
5	spr	一元运算符	6	
6	sqrt	一元运算符	6	
7	exp	一元运算符	6	
8	ln	一元运算符	6	自然对数
9	lg	一元运算符	6	以 2 为底的对数
10	sin	一元运算符	6	
11	cos	一元运算符	6	
12	arctan	一元运算符	6	
13	round	一元运算符	6	
14	trunc	一元运算符	6	
15	!	右一元运算符	6	阶乘
16	%	右一元运算符	6	百分比
17	+	二元运算符	4	
18	-	二元运算符	4	
19	*	二元运算符	5	
20	/	二元运算符	5	
21	^	二元运算符	6	
22	x	操作数	0.00000	
23	pi	操作数	3.14159	
24	e	操作数	2.71828	

图 13.8 表达式求值预定义的标记

在下面关于 Lexicon::set_standard_tokens 的实现中，我们初始化 String，让它包含所有的标准标记表，这些标记用空格分割。我们应用函数 get_word(**const** String &s, **int** n, String &t)；找到 String s 中的第 n 个词且将它写到 String t 中。这里的词定义为任何不包含空白的字符序列。用这个词命名的标记通过使用 Token 构造函数自动地加入到词典中。当然，这个标记构造函数调用 attributes 函数为新标记查找初始的数据记录。

```
void Lexicon::set_standard_tokens()
{
   int i=0;
   String symbols=(String)
     "; ()abs sqr sqrt exp ln lg sin cos arctan round trunc! % +-*/ ^x pi e";
   String word;
   while(get_word(symbols,i++,word)!=fail){
```

```
    Token t=word;
  }
  token_data[23].value=3.14159;
  token_data[24].value=2.71828;
}
```

1. 哈希表处理

我们需要设计用于计算下标的哈希函数。回忆一下，数组 Lexicon::token_data 取得一个下标并返回标记名和关于此标记的信息，哈希表 Lexicon::index_code 接受一个标记名且返回一个下标。

通过考虑哈希表所用的应用，经常可以增强哈希函数的性能。在我们的画图程序中，许多标记是单个的字符(一些是字母，一些是字符运算符)。我们设计的哈希函数因此特别强调了这一事实。

```
int Lexicon::hash(const String &identifier) const
/* Post: Returns the location in table Lexicon::index_code that corresponds to the
         String identifier. If the hash table is full and does not contain a record for
         identifier, the exit function is called to terminate the program.
   Uses: The class String, the function exit.*/
{
   int location;
   const char*convert=identifier.c_str();
   char first=convert[0],second; //  First two characters of identifier
   if(strlen(convert)>=2)second=convert[1];
   else second=first;
   location=first %hash_size;
   int probes=0;
   while(index_code[location]>=0 &&
         identifier!=token_data[index_code[location]].name){
     if(++probes> <=hash_size){
        cout<<" Fatal Error:Hash Table overflow.Increase its size\n";
        exit(1);
     }
     location+=second;
     location%=hash_size;
   }
   return location;
}
```

在这个函数中，通过在输出调试信息后，调用<cstdlib>中的系统函数 exit 去终止整个程序，以此响应哈希表的溢出。我们的程序不包含用户丢弃词典数据的任何方式，因此一旦发生溢出，不存在有用的恢复选项可用。如果我们希望升级这个程序，允许它从词典中删除，则最方便的响应溢出的办法是在哈希函数中抛出一个异常。①

13.5.5　表达式：标记表

*** 需要的方法

我们已决定用标记代码表去表示表达式(中缀和后缀式)，从而可以利用标准的表操作。

① 这本书中没有使用异常，但在高级 C++课本中有相关解释。

我们的表达式必须容许早先在这一章使用和开发的方法 get_token，infix_to_postfix，evaluate_postfix 和 recursive_evaluate，此外，我们当然还需要读、写、清除表示式以及计算表达式中标记数目的方法。

* * * 错误检查

为了有效地处理用户错误，有必要加入 valid_infix() 方法去检查一个中缀表达式是否在语句构成上是合理的。最后，由于 get_token 方法在表达式上不断向前移动，我们将需要一个 rewind 方法返回表达式的起始位置。当用户想用不同的图形边界或参数画同样的函数两次时需要调用这个方法。因此，Expression 类的说明具有如下形式：

```
class Expression {
public:
   Expression();
   Expression(const Expression &original);
   Error_code get_token(Token &next);
   void put_token(const Token &next);
   Expression infix_to_postfix();
   Error_code evaluate_postfix(Value &result);
   void read();
   void clear();
   void write();
   Error_code valid_infix();
   int size();
   void rewind();
private:
   List<Token>terms;
   int current_term;
   Error_code recursive_evaluate(const Token &first_token,
                                 Value &result, Token &final_token);
};
```

1. 处理标记表

get_token 方法在进行过程中从表达式中检索标记，为了在表达式上向前移，它需要递增数据成员 current_term。

```
Error_code Expression::get_token(Token &next)
/* Post: The Token next records the current_term of the Expression ; current_term is
         incremented; and an error code of success is returned, or if there is no such
         term a code of fail is returned.
   Uses: class List.*/
{
   if(terms.retrieve(current_term, next) != success) return fail;
   current_term++;
   return success;
}
```

类似地，put_token 方法只需要在表达式表的末尾插入一个标记，rewind 方法将 current_term 重置为 0。

2. 读取表达式

Expression::read 方法用于读取普通(中缀)形式的表达式，将它分割成标记，并将这些

标记的代码放入 Expression 中。

* * * 输入格式

现在必须建立有关输入格式的约定。我们假设每个输入表达式作为一行键入，则当我们到达这一行的行尾时，我们也到达输入字符串的结尾。我们使用 C++中关于空格的约定：忽略标记之间的空白，但空白的出现会终止一个标记。如果标记是一个词，则它应以字母开始，后面可以跟字母或数字。

因此为了读表达式，我们用 6.3 节中设计的 String 函数 read_in 读取一行文本。我们运用 add_spaces 函数在输入字符串 String 的每个运算符和分隔符的任一端插入空格；这样保证标记作为 String 的词出现。因此，我们能用 String 的函数 get_word 将文本分开。我们可以使用 Token 的构造函数将单个的词转换成标记。用这样的策略，我们得到下面的函数：

```
void Expression::read()
/* Post: A line of text, entered by the user, is split up into tokens and stored in the
        Expression.
   Uses: Classes String, Token, and List.*/
{
   String ,word;
   int term_count=0;
   int x;
   input=read in(cin,x);
   add_spaces(input);                    //  Tokens are now words of input.
   bool leading=true;
   for(int i=0; get_word(input,i,word)!=fail; i++){//  Process next token
      if(leading)
         if(word=="+")continue;                                  //  unary+
         else if(word=="-")word="~";                             //  unary-
      Token current=word;
                                         //  Castword to Token.
      terms.insert(term_count++,current);
      Token_type type=current.kind();
      if(type==leftparen||type==unaryop||type==binaryop)
         leading=true;
      else
        leading=false;
   }
}
```

方法 read 包含一部分需要进一步解释的代码，这与‘+’和‘-’两个符号有关，它们可以是一元运算符，也可以是二元运算符。我们引入布尔变量 leading 告之出现了哪种情况。

* * * leading 位置

leading 的值检测运算符是否有左参数。我们将说明如果 leading 为真，则当前标记可能没有左参数，因此不可能是合法的二元运算符。这样，我们的方法就能区分一元和二元的+和-运算符。对一元运算符‘+’，由于它没有影响，因此我们不采取动作。而对一元运算符‘-’，我们用专门定义的符号‘~’去替换。然而注意，这种修改只局部用于我们的程序，不要求——甚至不允许用户使用‘~’作为一元取反。

3. 前导和非前导位置

* * * 前导位置

为了说明包含变量 leading 的理由，我们首先考虑一种特殊情况。假设一个表达式只由

简单的操作数和二元运算符组成，没有括号或一元运算符，则惟一在语句构成上正确的表达式具有如下形式：

操作数 二元运算符 操作数 二元运算符 …… 操作数

这里第一个和最后一个标记都是操作数，并且这两类标记相互交替出现。两个操作数邻接或两个二元运算符邻接的情况都是非法的。前导位置必须是一个操作数，必须是在每个运算符后面，由于前面的运算符必定导致操作数，因此我们能将这些位置也看成“前导”位置。

现在假设向前面的表达式中插入一元运算符。在任何操作数前可放置任意数目的左一元运算符，但直接在二元运算符之前放置左一元运算符是非法的。也就是说，位于左侧的一元运算符正好能出现在允许操作数的地方，在前导位置且仅能在那里。另一方面，左一元运算符的出现使这个位置仍然是一个“前导”位置，因为操作数仍必须出现在二元运算符之前才是合法的。

	前一标记 任一种	合法的标记 任一种
前导位置：		
	表达式的开始	操作数
	二元运算符	一元运算符
	一元运算符	左括号
	左括号	
非前导位置：		
	操作数	二元运算符
	右一元运算符	右一元运算符
	右括号	右括号
		表达式的结束

图 13.9 前导和非前导位置中的合法标记

类似地，右一元运算符也能放在任何非前导位置（即在操作数或其他右一元运算符之后），右一元运算符的出现使这个位置成为非前导的，因为二元运算符必须出现在另一个操作数之前才是合法的。

最后，我们现在也允许表达式中的括号。用括号括起的子表达式作为操作数对待，因此，它们正好能出现在操作数是合法的那些地方。因此左括号正好能出现在前导位置并使这个位置成为前导的，而右括号仅能出现在非前导位置并使这个位置成为非前导的。

图 13.9 中概括了所有的可能。

4. 正确语法的错误检查

* * * 错误检查

在读入字符串的过程中，需要最大数量的错误检查以确信输入表达式的语法是正确的，从而使我们的程序尽可能健壮。这种错误检查在辅助方法 valid_infix 中完成，这个方法检查前导和非前导位置中合适种类的标记。下面的函数也检查括号正确地匹配：

```
Error_code Expression::valid_infix()
/* Post: A code of success or fail is returned according to whether the Expression is a
         valid or invalid infix sequence.
   Uses: Class Token.*/
{
   Token current;
   bool leading=true;
   int paren_count=0;
   while(get_token(current)!=fail){
     Token_type type=current.kind();
     if(type==rightparen||type==binaryop||type==rightunaryop){
       if(leading)return fail;
     }
     else if(!leading)return fail;
     if(type==leftparen)paren_count++;
     else if(type==rightparen){
       paren_count--;
       if(paren_count<0)return fail;
     }
     if(type==binaryop||type==unaryop||type==leftparen)
        leading=true;
     else leading=false;
   }
   if(leading)return fail;      // An expected final operand is missing.
   if(paren_count> <0)return fail; // Right parentheses are missing.
   rewind();
   return success;
}
```

5. 转换成后缀式

在 read()方法结束时,输入表达式已被转换成中缀的标记序列,以 13.4 节中导出的函数 infix_to_postfix 所需要的形式。实际上,我们现在到了算法的关键步骤,能应用先前的工作,仅对允许右一元运算符需做微小的修改。

当 infix_to_postfix 方法结束时,输出表达式是后缀形式的标记序列,在下一阶段即可高效地求出它的值。实际上,这种效率是重要的,这样即使需要用大量不同的值对表达式求值,也不用过度延迟便可画出图形。

6. 后缀求值

为求出后缀表达式的值,我们再次使用这一章第一部分中设计的方法。evaluate_postfix 方法的递归或非递归版本不用做显著的修改都可以使用。当然,evaluate_postfix 需要辅助函数 get_value,do_unary 和 do_binary,这些函数我们随后会介绍。

13.5.6　辅助的求值函数

在对后缀表达式求值时,我们需要辅助函数求出操作数的值及应用代表运算符的标记。

1. 操作数求值

get_value 函数仅需要调用 Token::value():

```
Value get_value(const Token &current)
/* Pre:  Token current is an operand.
   Post: The Value of current is returned.
   Uses: Methods of class Token.*/
{
   return current.value();
}
```

2. 运算符

由于我们有所有标记的整数代码，因此可以在一个简单而较长的 switch 语句中完成运算符的应用。我们将一元运算符留作练习。对二元运算符，我们有下面的函数：

```
Value do_binary(const Token &operation,
                const Value &first_argument, const Value &second_argument)
/* Pre:  Token operation is a binary operator.
   Post: The Value of operation applied to the pair of Value parameters is returned.
   Uses: Methods of class Token.*.
{
   switch(operation.code_number()) {
   case 17:
     return first_argument+second_argument;
   case 18:
     return first_argument-second_argument;
   case 19:
     return first_argument*second_argument;
   case 20:
     return first_argument/second_argument;
   case 21:
     return exp(first_argument,second_argument);
   }
}
```

* * * 求幂

求幂函数 exp(double)，由标准库<math.h>和<cmath>之一提供。

13.5.7 画出表达式的图形：Plot 类

最后，我们归结到整个程序的目的，即在计算机屏幕上画出表达式的图形。C++中的图形库完全依赖于系统，因此在一种机器上能运行的程序在另一种机器上可能不是必定能够运行。我们采用一种独立于系统的方法，使用普通的字符画出粗糙的图形。作为结果的类 Plot 能够简单地被利用系统图形能力的更高级的实现替换。之后，我们将详细说明这样一个增大的 Plot 类的实现，它适宜由 Borland C++[①]编译器使用。

1. Plot 类

Plot 对象中的数据成员用于存储图形的边界和要标绘的数据点。这些边界能简单地作

① Borland C++是 Borland International, Inc. 的商标。

为浮点数据成员 x_low，x_high，y_low 和 y_high 存储。另一个浮点数据成员 x_increment 设置相继数据点的 x 坐标之间的间隔。

当我们开始画图时，需要对要标绘的点进行排序，为此我们将这些点存储在一个 Sortable_list 中。因此，Plot 类如下说明：

```
class Plot {
public:
  Plot();
  Error_code set_limits();
  void find_points(Expression &postfix);
  void draw();
  void clear();
  int get_print_row(double y_value);
  int get_print_col(double x_value);
private:
  Sortable_list<Point>points; //  records of points to be plotted
  double x_low,x_high;        //  x limits
  double y_low,y_high;        //  y limits
  double x_increment;         //  increment for plotting
  int max_row,max_col;        //  screen size
};
```

find_points 方法从它的参数 Expression postfix 创建要标绘的点的 Sortable_list。

另一个重要的类方法是 draw，它利用存储在 Sortable_list 中的点数据，在用户屏幕上画出一个图。

Plot 类也带有构造函数、清除已存数据的方法 clear 以及在标绘特定点时用到的定位用户屏幕行列的方法。我们将输出屏幕看成矩形，它的尺寸由 Plot 的数据成员 max_row 和 max_col 确定。我们在 Plot 构造函数中指定这些数据成员。例如，我们将它们的值定为 19 和 79，从而得到 20×80 的文本屏幕，或者是 780 和 1024 的图形屏幕。屏幕上的点通过给出其行列坐标指定，行列坐标的范围是 0≤row≤max_row 和 0≤col≤max_col。get_print_row 方法使用图形上的点的 y 一坐标计算屏幕上相应的行。

```
int Plot ::get_print_row(double y_value)
/*Post: Returns the row of the screen at which a point with y- coordinate y_value
        shouldbe plotted, or returns -1 it the point would fall off the edge of the
        screen.*/
{
  double interp_y=
    ((double)max_row)*(y_high-y_value)/(y_high-y_low)+0.5;
  int answer=(int)interp_y;
  if(answer<0||answer>max row)answer=-1;
  return answer;
}
```

这个函数返回值 -1 指示来自图形边界之外的输入坐标。

2. 点

Plot 中的 Sortable_list 存储 Point 类的对象。由于这些对象代表要标绘的数据点，因此每个 Point 必须包含两个整型数据成员：row 和 col，它们确定了用户屏幕上的一个位置。这

些数据成员完全决定了点的次序，我们因此能简单地将 Point 看成它自己的有序键。在我们的程序中，这一决定用下面的定义实现：**typedef** Point Key;。

为确保与我们早先的排序方法兼容，每个 Point 必须有重载的比较运算符。我们现在定义的 Point 结构也包含两个构造函数，这些构造函数或者创建一个不带可用数据的 Point，或者创建一个存有给定行列参数值的 Point。

```
struct Point {
   int row,
   int col;
   Point();
   Point(int a, int b);
   bool operator==   (const Point &p);
   bool operator!=   (const Point &p);
   bool operator>=   (const Point &p);
   bool operator<=   (const Point &p);
   bool operator>    (const Point &p);
   bool operator<    (const Point &p);
};
```

最后，当我们在屏幕上标绘数据时，首先必须标绘属于屏幕顶部的那些数据点，然后再标绘较低的点。类似地，如果两个数据点位于屏幕的同一行，则我们将首先标绘最左边的一个点。因此我们要设计 Point 的比较运算符，使得必须及早标绘的点比稍后标绘的点要小。这样，我们对这些点进行简单的排序，按它们在用户屏幕上标绘的次序安排它们。例如，Point::**operator**<有如下实现：

```
bool Point::operator<(const Point &p)
{
return(row<p.row)||(col<p.col && row==p.row);
}
```

3. 创建 Point 数据

Plot::find_points 方法产生要标绘的点的列表，它必须重复地求出其后记表达式参数的值并将结果得到的点插入到 Sortable_list Plot::points 中。

```
void Plot::find_points(Expression &postfix)
/*Post: The Expression postfix is evaluated for values of x that range from x_low to x_
        high in steps of x_increment. For each evaluation we add a Point to the Sortable_
        list points.
  Uses: Classes Token, Expression, Point, and List.*/
{
  double x_val=x_low;
  double y_val;
  while(x_val<=x_high){
    Token::set_x(x_val);
    postfix.evaluate_postfix(y_val);
    postfix.rewind();
    Point p(get_print_row(y_val),get_print_col(x_val));
    points.insert(0,p);
    x_val+=x_increment;
  }
}
```

4. 画出图形

一旦我们生成了点的 Sortable_list，就准备画图了。我们简单地对这些点进行排序，在我们的实现中是采用归并排序，然后遍历整个有序表，在每个指示的屏幕位置放置一个符号。我们首先对要标绘的点的列表进行排序以保证它们在图形输出中已经是自顶向下排列，而在任何给定的高度，又是从左到右排列的。

```
void Plot::draw()
/* Post: All screen locations represented in points have been marked with the
         character '#' to produce a picture of the stored graph.
  Uses: Classes Point and Sortable_list and its method merge_sort.*/
{
  points.merge_sort();
  int at_row=0,at_col=0; //  cursor coordinates on screen
  for(int i=0; i<points.size(); i++){
     Point q;
     points.retrieve(i,q);
     if(q.row<0||q.col<0)continue; //  off the scale of the graph
     if(q.row<at_row||(q.row==at_row && q.col<at_col))continue;
     if(q.row>at_row){                    //  Move cursor down the screen.
        at_col=0;
        while(q.row>at_row){
          cout<<endl;
          at_row++;
        }
     }
     if(q.col> at_col)                    //  Advance cursor horizontally.
       while(q.col>at_col){
         cout<<" ";
         at_col++;
       }
     cout<<"#";
     at_col++;
  }
  while(at_row++<=max_row)cout<<endl;
}
```

13.5.8　增强的图形 Plot 类

虽然在 C++ 的标准库中没有图形支持，但这种支持经常由特定的系统提供。例如，Borland C++编译器在头文件<graphics. h>中包含有图形例程库，这个库包括初始化图形屏幕、检测屏幕像素数、标记单个像素和恢复屏幕的函数。

我们可以通过仅修改与输出屏幕有某些交互影响的 4 个函数的实现，将这些例程合并到自己的 Plot 类中。这些方法是构造函数和操作 draw，find_points 和 get_print_col。

在对构造函数进行非常微小的修改中，我们要移去 max_row 和 max_col 上预定义的限制，因为它们现在是由库函数计算。在 get_print_col 方法中也需要做类似微小的修改，以反映像素行是自屏幕的底部向上，而不是自屏幕的顶部向下。

在 find_points 方法中，我们从使用 Borland 的图形函数初始化屏幕开始且记下它的尺寸，然后列出输出数据点，方式同早先的实现。

```
void Plot::find_points(Expression &postfix)
/*Post: The Expression postfix is evaluated for values of x that range from x_low to x_
        high in steps of x_increment. For each evaluation we add a Point of the
        corresponding data point to the Sortable_list points.
  Uses: ClassesToken, Expression, Point, and List and the library<graphics.h> .*/
{
  int graphicdriver=DETECT, graphicmode;
  initgraph( &graphicdriver, &graphicmode, "");//  screen detection and
  graphresult();                                //  initialization
  max_col=getmaxx();                            //  with graphics.h library
  max_row=getmaxy();
  double x_val=x_low;
  double y_val;
  while(x_val<=x_high){
    Token::set_x(x_val);
    postfix.evaluate_postfix(y_val);
    postfix.rewind();
    Point p(get_print_row(y_val),get_print_col(x_val));
    points.insert(0,p);
    x_val+=x_increment;
  }
}
```

最后，draw 方法没有必要排序点，因为像素能以任何次序标绘。这个方法简单地列出点并标记相应的像素。

```
void Plot::draw()
/*Post :All pixels represented in points have been marked to produce a picture of the
       stored graph.
  Uses: Classes Point and Sortable_list and the library<graphics.h>*/
{
  for (int i=0; i<points.size(); i++){
      Point q;
      points.retrieve(i,q);
      if(q.row<0||q.col<0)continue; //  off the scale of the graph
      if(q.row><max_row||q.col><max_col)continue;
      putpixel(q.col,q.row,3);       //  graphics.h library function
    }
    cout "Enter a character to clear screen and continue:"<<flush;
    char wait_for;
    cin>>wait_for;
    restorecrtmode();                //  graphics.h library function
}
```

至此，我们已通盘考虑了整个项目。在课本中还剩下几个函数和方法尚未出现，但这些函数和方法都非常简单易懂，因此将它们都留作练习。

练习 13.5

E1. 准确地说明加入以 10 为底的对数函数 log()作为附加的一元运算符时在程序中需要做

何修改。

E2. 本程序的无知的用户可能(如果画出涉及金钱的函数图形)在表达式中写美元符号‘$’。如果这样的话，当前的程序将会做什么？需要对此程序做何修改以使它忽略‘$’？

E3. C++程序设计员可能在表达式结尾偶然地键入一个分号‘;’。要忽略表达式结尾处的分号，但视其他地方出现的分号为错误，需要做哪些修改？

E4. 为了允许程序像接受圆括号(…)一样接受方括号[…]或花括号{…}，需要做何修改。嵌套必须用同样种类的括号进行，即具有(…[…)…]形式的表达式是非法的，而像[…(…)…{…}…]这样的形式是允许的。

程序设计项目 13.5

P1. 提供缺失的函数和方法，并在你的计算机上实现画图程序。

进阶参考书目

波兰表示法是如此自然有用，有人或许会认为它的发现是几百年前的事，但令人吃惊的是，它是本世纪的发现：Jan Łukasiewicz, *Elementy Logiki Matematyczny*, Warsaw, 1929; 英文译本：*Elements of Mathematical Logic*, Pergamon Press, 1963.

形成和计算波兰表达式(一般是后缀形式)的迭代算法的设计可在几本数据结构书和关于编译器理论的更高级的书中找到。将表达式从中缀式转换成后缀式的迭代算法，E. W. Dijkstra 和 C. L. Hamblin 各有独立的见解：E. W. Dijkstra, “Making a translator for ALGOL 60”, *Automatic Programming Information number* 7(May 1961)；再版于 *Annual Revue of Automatic Programming* 3(1963), 347-356. C. L. Hamblin, “Translation to and from Polish notation”, *Computer Journal* 5 (1962), 210-213.

后缀表达式求值的递归算法来源虽然有相当不同的观点，但仅对二元运算符来说，是 Edward M. Reingold, “A comment on the evaluation of Polish postfix expressions”, *Computer Journal* 24(1981), 288.

附录 A　数学方法

此附录的第一部分提供算法分析中使用的一些数学结论，附录的最后两节是斐波纳契数和 Catalan 数，这两节是可选的，主题面向爱好数学的读者。

A.1　整数幂的和

下面的两个公式在计算算法执行步骤时很有用。

定理 A.1

$$1+2+\cdots+n=\frac{n(n+1)}{2}$$

$$1^2+2^2+\cdots+n^2=\frac{n(n+1)(2n+1)}{6}$$

证明　第一个等式有简单巧妙的证明。设 S 等于左边的和，将它书写两遍（每遍各按一个方向），并垂直相加：

$$\begin{array}{ccccccccccc} 1 & + & 2 & + & 3 & + & \cdots & + & n-1 & + & n & = S \\ n & + & n-1 & + & n-2 & + & \cdots & + & 2 & + & 1 & = S \\ \overline{n+1} & + & \overline{n+1} & + & \overline{n+1} & + & \cdots & + & \overline{n+1} & + & \overline{n+1} & = \overline{2S} \end{array}$$

左边有 n 列，因此 $n(n+1)=2S$，于是公式成立。（**证明结束**）

第一个恒等式也可以如图 A.1 所示不用任何语句进行证明。

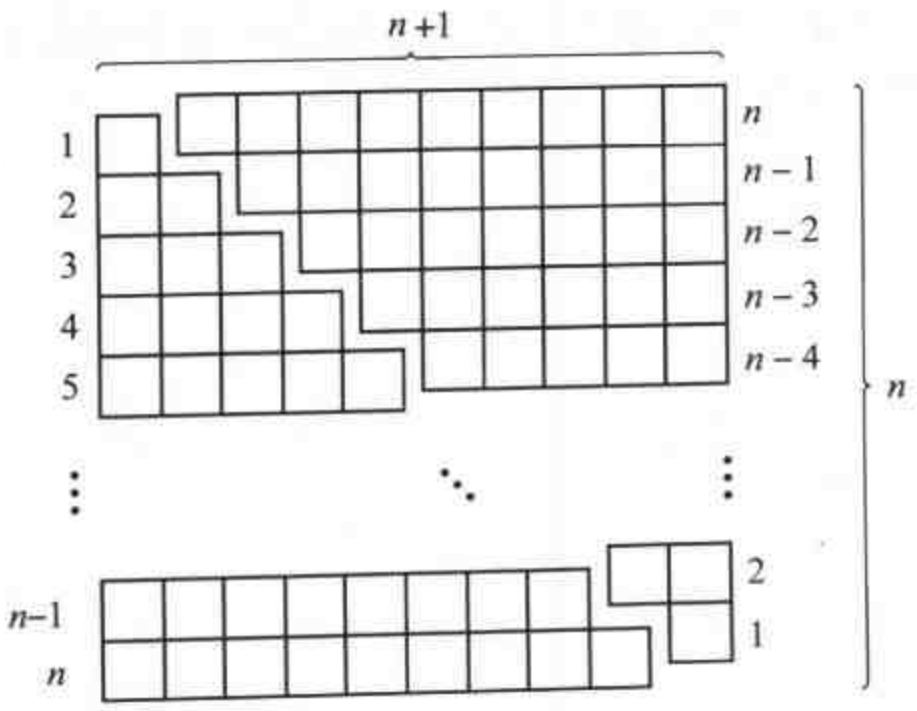


图 A.1　整数和的几何证明

＊＊＊用递归证明

我们将使用数学归纳法证明第二个恒等式，此方法要求从建立一个初始情况开始，我们称之为归纳基础，对我们的公式来说初始情况是 $n=1$，此时公式变成：

$$1^2=\frac{1(1+1)(2+1)}{6},$$

它是成立的，因此建立了归纳基础。接下来，我们使用 $n-1$ 时的公式去建立 n 时的公式。对 $n-1$，我们假设：

$$1^2+2^2+\cdots+(n-1)^2=\frac{(n-1)n(2(n-1)+1)}{6}$$

随后得到：

$$\begin{aligned}1^2+2^2+\cdots+(n-1)^2+n^2&=\frac{(n-1)n(2(n-1)+1)}{6}+n^2\\&=\frac{2n^3-3n^2+n+6n^2}{6}\\&=\frac{n(n+1)(2n+1)}{6}\end{aligned}$$

它正是期望的结果，因此归纳证明完成。

＊＊＊连加记号

这些恒等式中出现的这种求和的一种方便的简写是在典型的被加数之前使用大写的希腊字母西格玛 $\sum$ 将被加数的下标初值写在此符号的下面，其终值写在符号的上面，于是前面的恒等式能够写成：$\sum_{k=1}^{n}k=\frac{n(n+1)}{2}$ 和 $\sum_{k=1}^{n}k^2=\frac{n(n+1)(2n+1)}{6}$。

另外两个公式也很有用，特别是在处理关于树的问题时。

定理 A.2

$$1+2+4+\cdots+2^{m-1}=2^m-1$$

$$1\times1+2\times2+3\times4+\cdots+m\times2^{m-1}=(m-1)\times2^m+1$$

采用连加记号，这两个等式可以写成：

$$\sum_{k=0}^{m-1}2^k=2^m-1$$

$$\sum_{k=1}^{m}k\times2^{k-1}=(m-1)\times2^m+1$$

证明 我们将用更通用的形式证明第一个公式。我们从下面的恒等式开始，对 $x\neq1$ 的任意值，通过在两边都乘上 $x-1$，能够证明：

$$\frac{x^m-1}{x-1}=1+x+x^2+\cdots+x^{m-1}$$

当 $x=2$ 时，这个表达式就变成第一个公式。

为了建立第二个公式，我们对同样的表达式取 $m+1$ 而非 m 时的情况：对任意 $x\neq1$，

$$\frac{x^{m+1}-1}{x-1}=1+x+x^2+\cdots+x^m$$

再对上式关于 x 求导：对任意 $x\neq 1$，

$$\frac{(x-1)(m+1)x^m-(x^{m+1}-1)}{(x-1)^2}=1+2x+3x^2+\cdots+mx^{m-1}$$

置上式中的 $x=2$，则得到第二个公式。

假设在上面的公式中 $|x|<1$。当 m 变大时，x^m 变小，即 $\lim_{m\to\infty}x^m=0$。当 $m\to\infty$ 时，取上面等式的极限，我们得到：

* * * 无限级数

定理 A.3 如果 $|x|<1$，则：

$$\sum_{k=0}^{\infty}x^k=\frac{1}{1-x}\text{ 及 }\sum_{k=0}^{\infty}kx^{k-1}=\frac{1}{(1-x)^2}$$

A.2 对数

使用对数的主要原因是将乘法和除法转换成加法和减法，以及将指数转换成乘法。在便携计算器出现之前，对数是手工计算的一种必不可少的工具：大型对数表和曾经普遍存在的计算尺即是证明。尽管我们现在有数值计算的其他方法，对数的基本属性已经扩展到远远超过用作计算工具，因此仍显示出其重要性。

* * * 物理度量

首先，许多现象的特性反映了内在的对数结构，也就是说，通过使用对数，我们能发现重要的关系，而这种关系在通常情况下并不明显。例如，度量声音的响度是对数的：如果一种声音比另一种声音高 10dB(分贝)，则实际的声能超过 10 倍。如果一个房间内的声级是 40dB，而另一个房间内是 60dB，则人类的感觉可能是第二个房间的噪声是第一个房间的一半，但在第二个房间内实际上是 100 倍的声能。这种现象就是为什么单个小提琴独奏者在整个交响乐团中能被听见的原因(当以不同的乐谱线演奏时)，然而交响乐团要求许多小提琴以保持声音的适当平衡。

地震强度也按对数度量：每增加 1 里氏震级就表示释放出 10 倍的能量。

* * * 大数

其次，对数提供了方便的方法去处理较大的数。在科学表示法中，经常将一个数写成一个小的实数(通常在 1 到 10 的范围之间)乘以 10 的幂，它实际上是基于对数，因为 10 的幂本质上是这个数的对数。需要使用非常大的数的科学家(如天文学家、核物理学家和地质学家)经常谈到数量级，因而着重于这个数的对数。

最后，对数图是在比线性图更宽的范围内显示函数性质的非常有用的工具，用对数图，我们能安排显示自变量取较小值时函数的详细信息，同时也能给出较大值时的函数总体视图。当我们希望显示函数中的百分比变动时，对数图特别合适。

A.2.1　对数的定义

＊＊＊底数

对数按照实数 $a>1$ 进行定义，它称为对数的底数。（也可能用 $0<a<1$ 之间的底数 a 去定义对数，但这样做会在我们的讨论中引入不必要的复杂性。）对任何数 $x>0$，我们定义 $\log_a x=y$，这里 y 是实数，$a^y=x$。负数的对数和 0 的对数均不予定义。

A.2.2　简单的性质

从这个定义和指数的性质，我们得到：

$$\log_a 1=0$$
$$\log_a a=1$$
$$\log_a x<0, \text{对所有 } 0<x<1 \text{ 的 } x$$
$$0<\log_a x<1, \text{对所有 } 1<x<a \text{ 的 } x$$
$$\log_a x>1, \text{对所有 } a<x \text{ 的 } x$$

对数函数具有如图 A.2 中显示的图形。

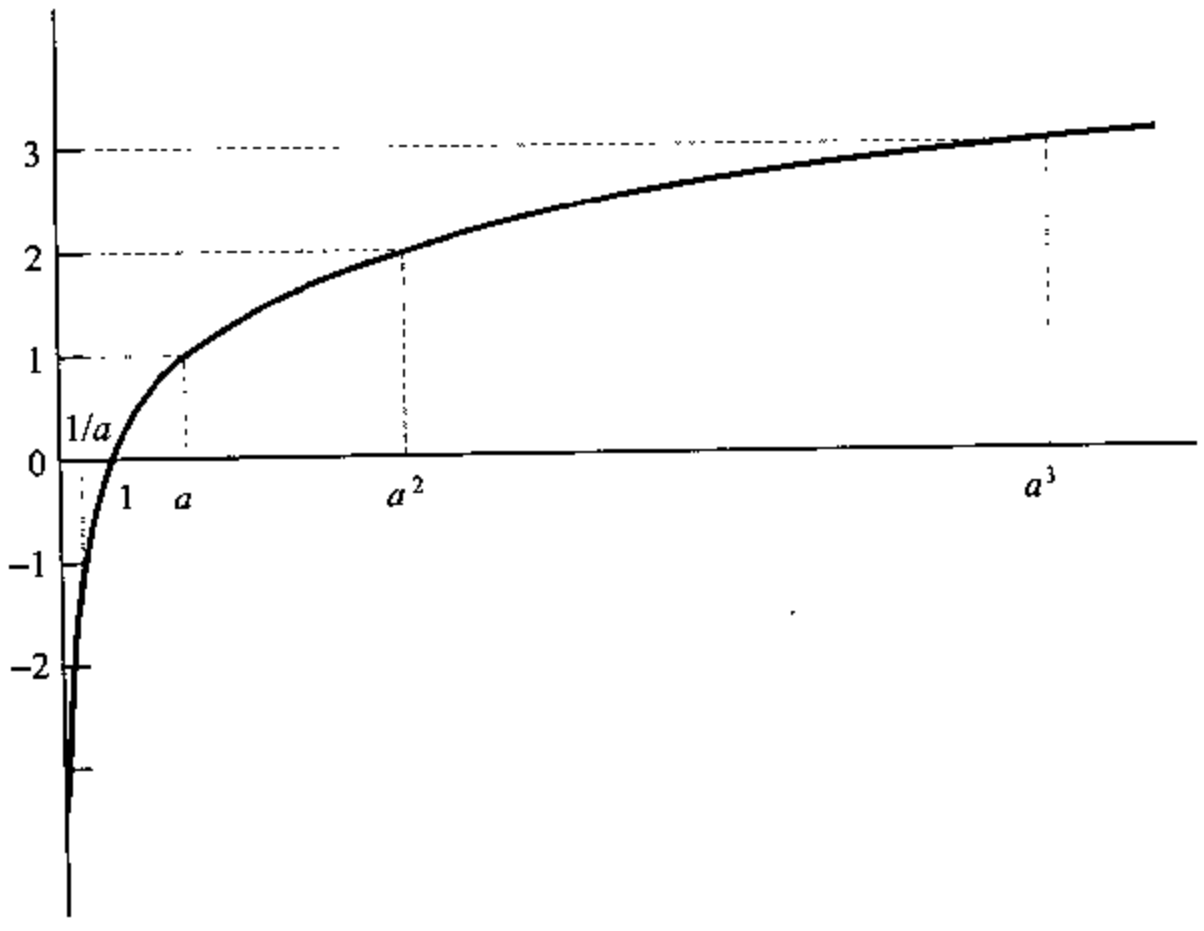


图 A.2　对数函数的图形

＊＊＊恒等式

我们也得到下面的恒等式：

$$\log_a(xy)=(\log_a x)+(\log_a y)$$
$$\log_a(x/y)=(\log_a x)-(\log_a y)$$
$$\log_a x^z=z\log_a x$$

$$\log_a a^z = z$$

$$a^{\log_a x} = x$$

对任何正实数 x 和 y 及任何实数 z，上列恒等式成立。

从图 A.2 中可以看出，随着 x 的增长，对数的增长越来越慢。小于 1 的 x 的正幂(如 x 的平方根或 x 的立方根)的图形，增长也变得越来越慢，但它们决不会像对数图形那样平缓。实际上，

当 x 增大时，对任何 $c>0$，$\log x$ 比 x^c 增长更加缓慢。

A.2.3 底数的选择

＊＊＊常用对数

任何实数 $a>1$ 均能被选作对数的底数，但某些特殊的选择比其他选择更经常使用。在计算和画图时，经常使用底数 $a=10$，底数为 10 的对数称为常用对数(common logarithm)。然而在研究计算机算法时，底数 10 并不经常出现，我们通常不使用常用对数，相反，底数为 2 的对数最经常出现，因此我们保留特定的符号 $\lg x$ 去表示底数为 2 的对数。

A.2.4 自然对数

＊＊＊自然对数

在研究对数的数学性质时，以及在部分答案中出现对数的许多问题中，作为底数的数是 $e=2.7182818284590\ldots$，底数为 e 的对数称为自然对数(natural logarithm)。在本书中经常将 x 的自然对数表示成 $\ln x$。然而在许多数学书中，很少使用除 e 之外其他的底数，在此情

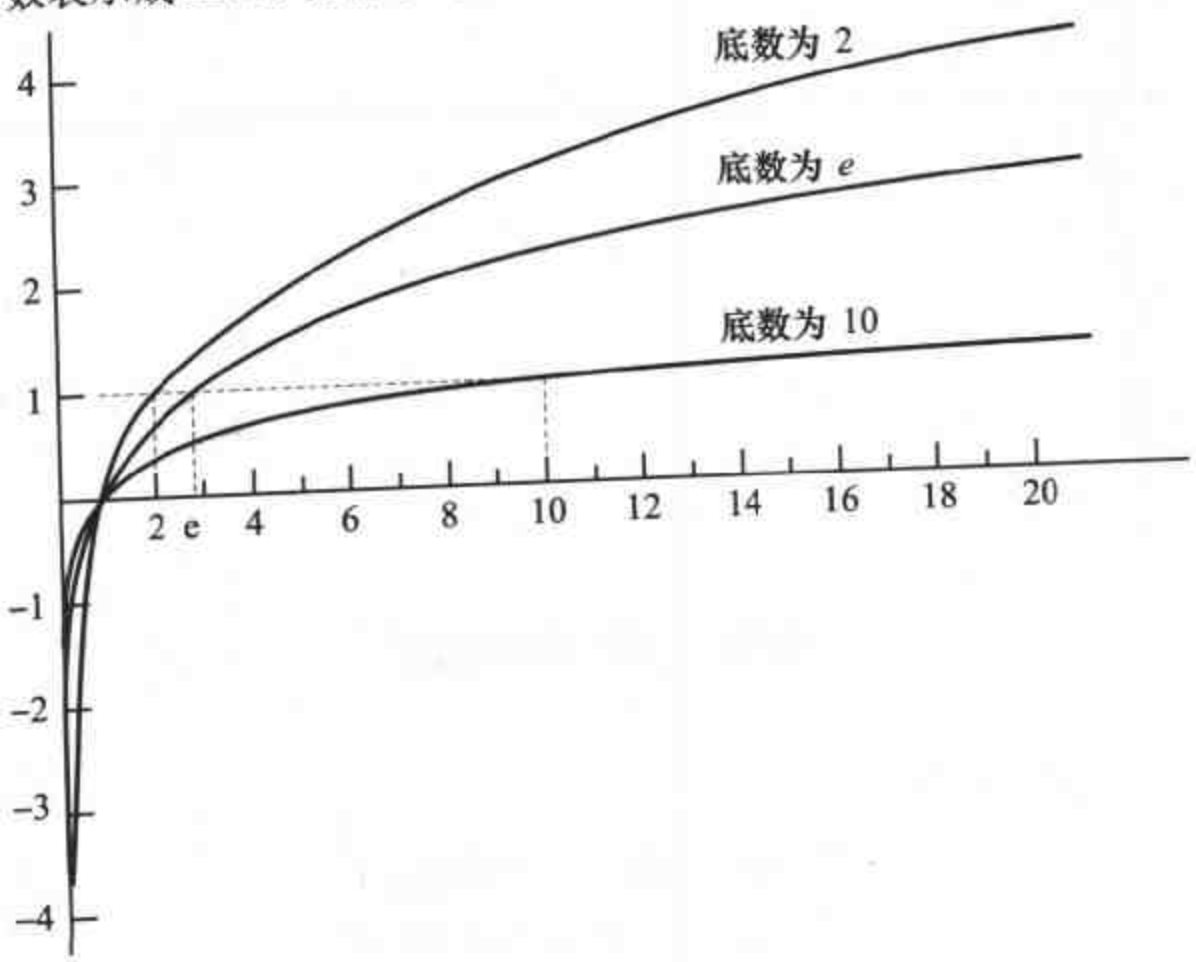


图 A.3 3 种底数的对数图

况下，未加限制的符号 $\log x$ 一般用于表示自然对数。图 A.3 显示了分别以 2，e 和 10 为底的对数图。

＊＊＊指数函数

使 e 成为底数之自然选择的对数性质完全是从微积分学的一部分发展而来，但我们可以提出其中的两条性质而不用证明。首先，$\ln x$ 的图形具有在每个点 x 的斜率为 $1/x$ 的性质，即对所有的实数 $x>0$，$\ln x$ 的导数是 $1/x$。其次，对 $-1<x<1$，自然对数满足无限级数：

$$\ln(x+1)=x-\frac{x^2}{2}+\frac{x^3}{3}-\frac{x^4}{4}+\cdots$$

但这个级数需要许多项给出较好的近似，因此它对计算不是直接有用。因此对所有实数 x，更好的考虑是消除了对数的指数函数，它满足级数：

$$e^x=1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\cdots$$

这个指数函数 e^x 也有重要的性质，它是自己的导数。

A.2.5 符号

刚才对不同底数的对数使用的符号将作为我们自己的标准，概括如下：

约 定

除非特别说明，否则所有的对数都以 2 为底数。

符号 lg 表示以 2 为底数的对数，

符号 ln 表示自然对数。

如果对数的底数没有指定或没有影响，则使用符号 log。

A.2.6 底的变化

具有一种底数的对数与具有另一种底数的对数密切相关，为找出这种关系，我们从下面的关系开始，本质上对任何 $x>0$，$x=a^{\log_a x}$，则

$$\log_b x=\log_b a^{\log_a x}=(\log_a x)(\log_b a)$$

因子 $\log_b a$ 不依赖于 x，而仅依赖于两个底数。因此：

为将对数从一种底数转换成另一种底数，需要乘上一个常数因子，这个因子是关于第二个底数的第一个底数的对数。

＊＊＊转换因子

就此而论，对我们最有用的数是：

$$\lg e \approx 1.442695041$$

$$\ln 2 \approx 0.693147181$$

$$\ln 10 \approx 2.302585093$$

$$\lg 1000 \approx 10.0$$

最后一个值是重要的近似 $2^{10}=1024 \approx 10^3=1000$ 的结果。

A.2.7 对数图

在对数刻度中，数的安排就像在计算尺上一样，大数比小数靠得更近。这样，刻度上的相等距离代表相等的比率，而不是普通线性刻度上代表的相等的差额。在度量百分比变化显得比较重要时，或者感觉上是对数时，就应当使用对数刻度。例如，有时人类对时间的感觉从短期观点来看好像是接近线性的——两天前发生的事情与昨天发生的事情距离两倍——但从长远观点来看更接近对数形式：我们很少在百万年前与两百万年前和十年与一百年前之间做出区分。

* * * 重对数图

水平和垂直刻度都是对数的图形称为重对数图(log-log graph)。除了两个刻度上感觉是自然对数的现象之外，重对数图在显示大范围的函数特性时很有用。对较小的值，重对数图记录了这个函数的详细视图，而对较大的值，这个函数的较宽范围的视图可以出现在同样

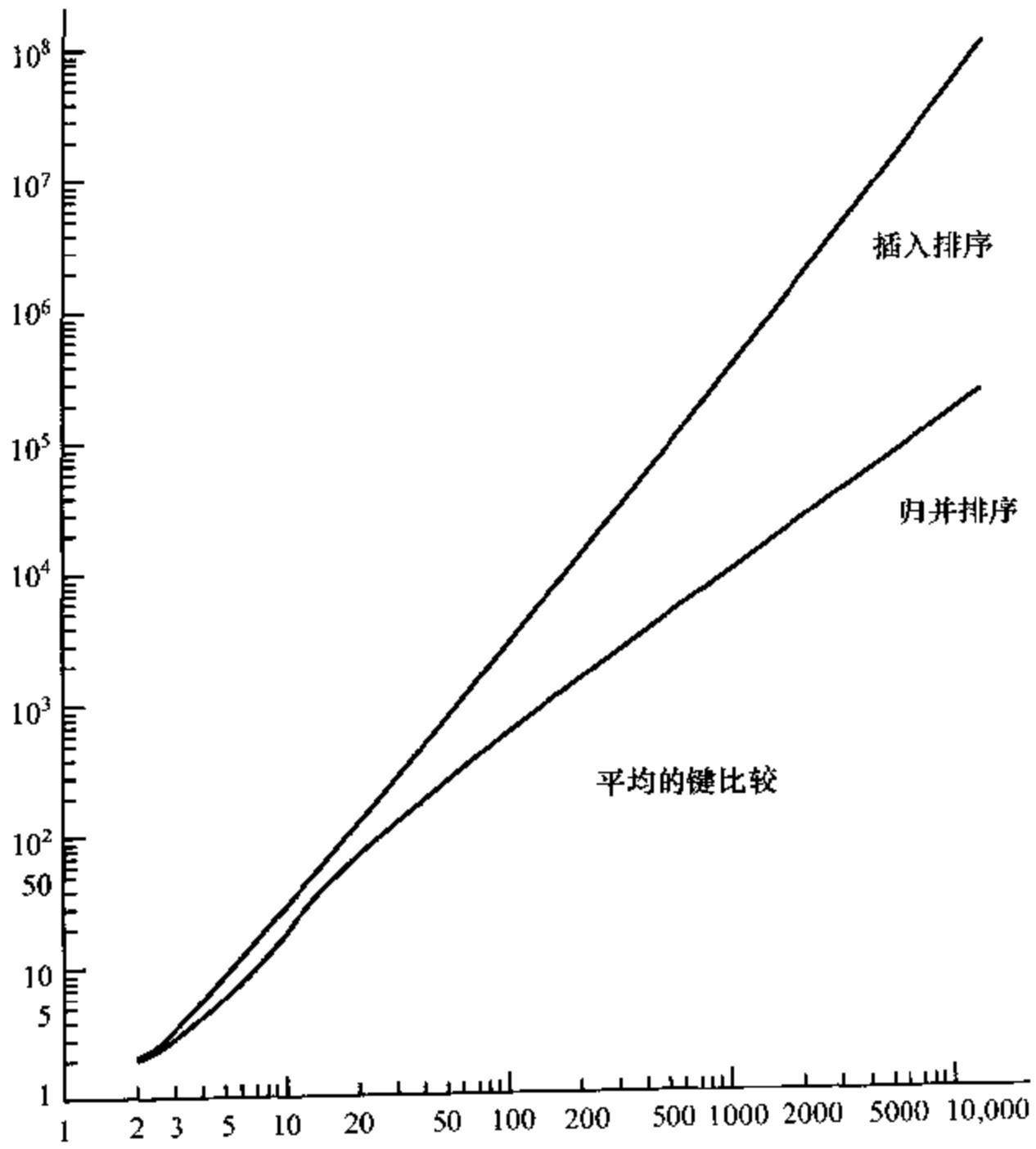


图 A.4 插入排序和归并排序比较次数的重对数图

的图上。对查找和排序算法，我们希望比较各种方法对小型问题和大型问题的情况，因此重对数图就显得比较合适。（见图 A.4。）

有一个观察结果值得提及：x 的任何次幂在重对数刻度上的图形是一条直线。为证明这一点，我们从任意的幂函数 $y=x^n$ 开始，并对其两边取对数，得到 $\log y=n\log x$。在 x 和 y 中的重对数图成为 $u=\log x$ 和 $v=\log y$ 中的线性图形，就 u 和 v 来说等式就变成 $v=nu$，它的图形确实是一条直线。

A.2.8　调和数

作为对数的最后一个应用，我们要得到算法分析中，特别是排序方法中经常出现的一个和的近似值。第 n 个调和数（harmonic number）定义为从 1 到 n 的整数的倒数之和：

$$H_n=1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}$$

为了计算 H_n，我们考虑函数 $1/x$ 和图 A.5 中所示的关系。阶跃函数（step function）下的面积明显是 H_n，因为每个阶跃的宽度为 1，而对从 1 到 n 的每个整数 k，第 k 个阶跃的高度为 $1/k$。我们将这个面积近似成曲线 $1/x$ 下从 $\frac{1}{2}$ 到 $n+\frac{1}{2}$ 之间的面积。此曲线下的面积是：

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}}\frac{1}{x}dx=\ln(n+\frac{1}{2})-\ln\frac{1}{2}\approx\ln n+0.7$$

当 n 较大时，小数项 0.7 无关紧要，因此我们将 $\ln n$ 作为 H_n 的较好的近似。

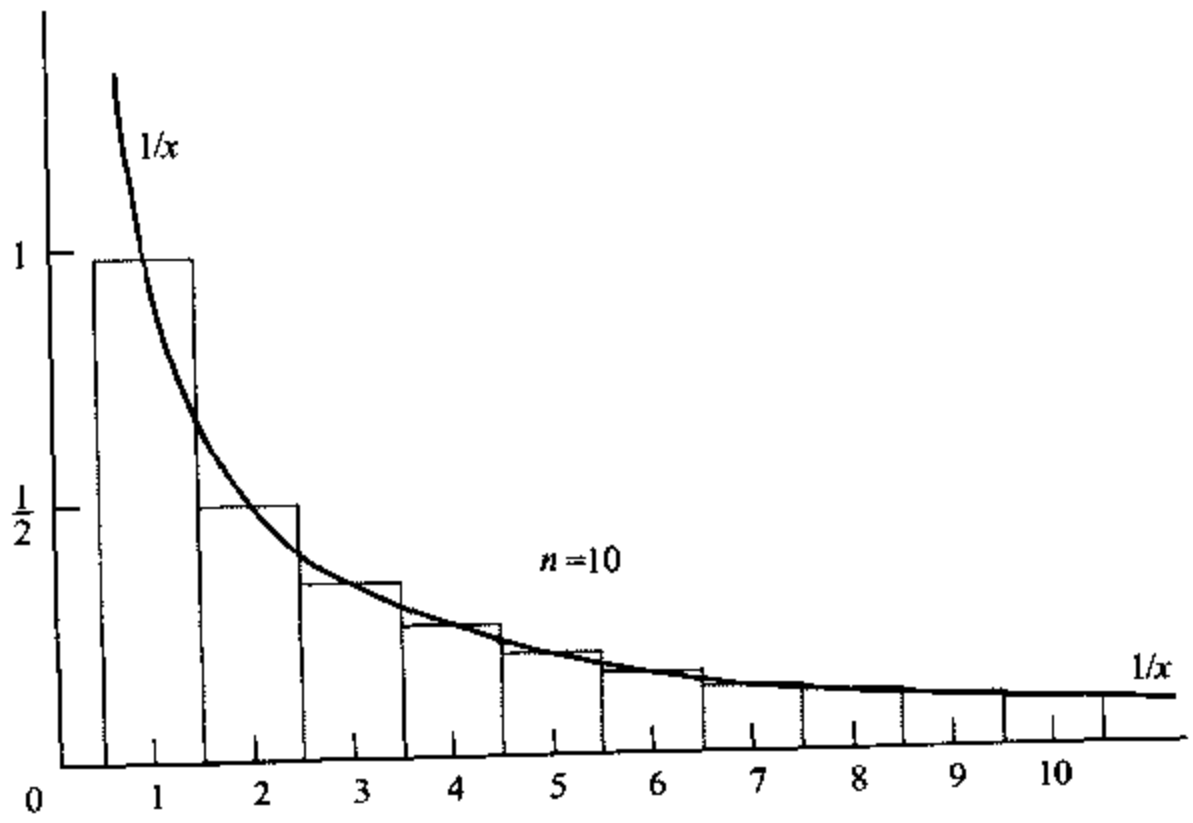


图 A.5　$\int_{\frac{1}{2}}^{n+\frac{1}{2}}\frac{1}{x}dx$ 的近似值

如果需要，通过改进这种用积分计算近似值的方法，则可能得到更接近于 H_n 的近似

值。特别地，

定理 A.4 当 $n \geqslant 1$ 时，调和数 H_n 满足：

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \epsilon$$

其中 $0 < \epsilon < 1/(252n^6)$，且 $\gamma \approx 0.577215665$，$\gamma$ 称为欧拉常数(Euler's constant)。

A.3 排列、组合和阶乘

A.3.1 排列

* * * 排列数

对象的排列(permutation)是这些对象的连续次序或安排。如果我们首先从 n 个不同的对象开始，则可以将其中的任一个对象选为此安排中的第一个，然后对第二个对象来说存在 $n-1$ 种选择，由于这些选择能以所有可能的方式进行组合，因此将各种选择的数目相乘。因而前两个对象可以用 $n(n-1)$ 种方式选择。现在还剩下 $n-2$ 个对象，其中的任何一个都可选为这个安排中的第 3 个。按此方式继续下去，我们发现 n 个不同对象的排列数是：

$$n! = n \times (n-1) \times (n-2) \times \cdots\cdots \times 2 \times 1$$

排列的对象	abcd					
首先选择 a	abcd	abdc	acbd	acdb	adbc	adcb
首先选择 b	bacd	badc	bcad	bcda	bdac	bdca
首先选择 c	cabd	cadb	cbad	cbda	cdab	cdba
首先选择 d	dabc	dacb	dbac	dbca	dcab	dcba

图 A.6 构造排列

注意我们已假设所有的对象完全不同，即我们可以区分每个对象。对不同对象的配置计数通常比有些对象不能区别时的情况更加简单。对后一个问题，有时通过将这些对象临时打上标记以使它们看上去互不相同，然后再对这些配置计数，最后再除以可以打标记的方式数。下一节中的特殊情况极其重要。

A.3.2 组合

从 n 个对象中每次取出 k 个对象的"组合"是从 n 个对象选出 k 个对象，其中不考虑选择的次序。这个组合数可以表示成：

$$C(n,k) \text{ 或 } \binom{n}{k}$$

* * * 组合数

我们可以从 n 个对象的 $n!$ 的排列开始计算 $C(n,k)$，并简单地通过选择排列中的前 k 个对象形成组合。然而，在确定组合时，将忽略这 k 个对象出现的次序，因此我们必须除以排列所选择的 k 个对象的 $k!$ 种方式。同时也忽略未被选择的其余 $n-k$ 个对象的次序，因

此我们也必须除以$(n-k)!$。因此：

$$C(n,k)=\frac{n!}{k!\ (n-k)!}$$

供选择的对象	abcdef			
abc	acd	adf	bcf	cde
abd	ace	aef	bde	cdf
abe	acf	bcd	bdf	def
adf	ade	bce	bef	def

图 A.7 从6个对象中每次取3个对象的组合

* * * 二项式系数

组合数$C(n,k)$称为二项式系数，因为它作为$(x+y)^n$的展开式中$x^k y^{n-k}$的系数出现。关于二项式系数的各种和与积有几百种不同的关系式和恒等式，其中最重要的一些可以在初等代数和组合数学的课本中找到。

A.3.3 阶乘

我们在分析算法时经常用到排列和组合，而对这些应用，我们必须估计n取各种不同值时$n!$的大小。James Stirling在18世纪给出了$n!$的一个极好的近似值：

定理 A.5

$$n!\approx\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left[1+\frac{1}{12n}+O\left(\frac{1}{n^2}\right)\right]$$

* * * Stirling近似

我们经常以对数形式使用这个近似值：

推论 A.6

$$\ln n!\approx\left(n+\frac{1}{2}\right)\ln n-n+\frac{1}{2}\ln(2\pi)+\frac{1}{12n}+O\left(\frac{1}{n^2}\right)$$

注意，随着n的增长，这个近似值的对数变得越来越精确，即它们的差接近于0。在阶乘的直接近似值和$n!$本身之间的差不必变小(即这个差不必趋于0)，但百分比误差变得任意小(此比率趋于1)。Knuth(第1卷，第111页)给出了Stirling近似值的精化，它更加接近$n!$。

证明 Stirling近似的完整证明需要高级微积分中的技巧，这将使我们远离此处的证明。然而我们能够使用一点初等微积分去说明此近似的第一步。首先，我们取一个阶乘的自然对数，注意积的对数是对数的和：

$$\begin{aligned}\ln n! &= \ln(n\times(n-1)\times\cdots\times 1)\\ &= \ln n+\ln(n-1)+\cdots+\ln 1\end{aligned}$$

$$= \sum_{x=1}^{n} \ln x$$

接下来，我们用积分估计这个和，如图 A.8 中所示。

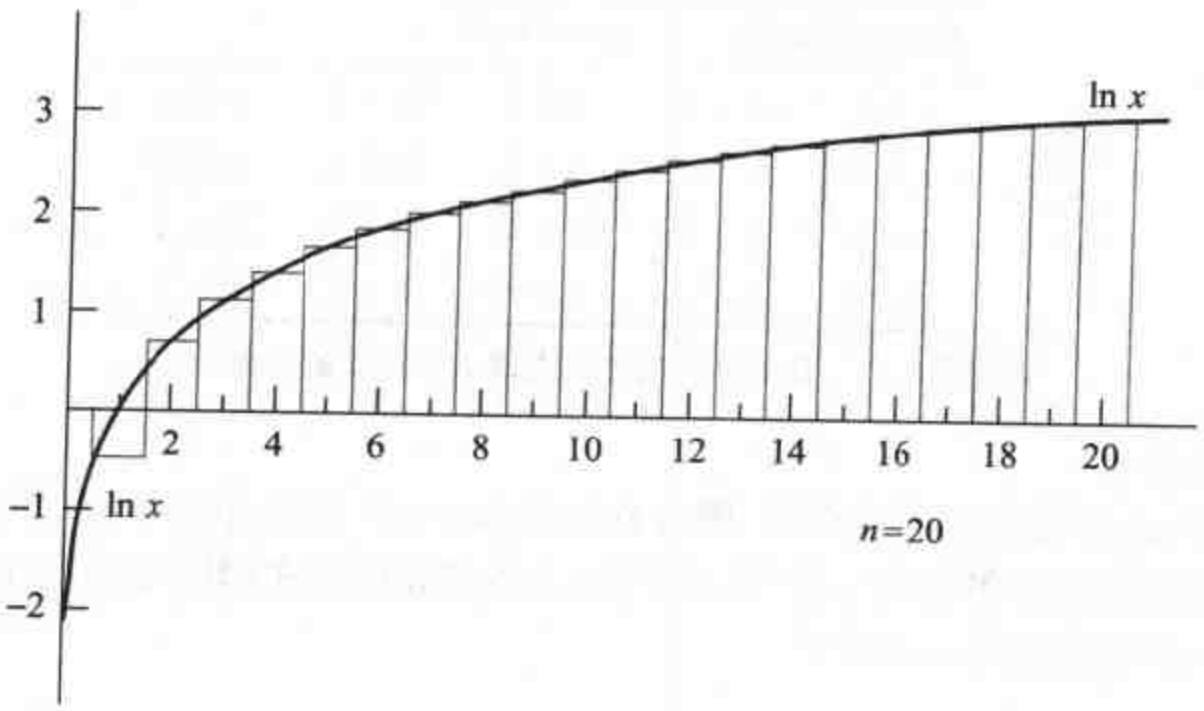


图 A.8　用 $\int_{\frac{1}{2}}^{n+\frac{1}{2}} \ln x dx$ 计算 $\ln n!$ 的近似值

从这个图中能明显地看出，阶跃函数下的面积正是 $\ln n!$，它与曲线下面的面积近似相等，这个面积是：

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \ln x dx = (x\ln x - x)\Big|_{\frac{1}{2}}^{n+\frac{1}{2}}$$

$$= (n+\frac{1}{2})\ln(n+\frac{1}{2}) - n + \frac{1}{2}\ln 2$$

对较大的 n 值，$\ln n$ 和 $\ln(n+\frac{1}{2})$ 之间的差别微不足道，因此这个近似值与 Stirling 近似值之间的差别仅是 $\frac{1}{2}\ln 2$（大约 0.35）和 $\frac{1}{2}\ln(2\pi)$（大约 0.919）之间的常数差。（**证明结束**）

A.4　斐波纳契数

斐波纳契数的起源是 Leonardo Fibonacci 在 1202 年提出的一道算术习题：

* * * 兔子

从一对兔子开始，在一年时间内能发展到多少对兔子？我们从新生的一对兔子开始，它们需要一个月的时间达到成熟，之后它们每个月生产一对新兔子，而且兔子永不死亡。

* * * 递推关系

在第 1 个月，我们只有一对兔子。在第 2 个月，我们仍然只有一对兔子，但它们现在已达到成熟。在第 3 个月，它们已经繁殖，因此现在有两对兔子。于是继续下去，在第 n 个月

所拥有的兔子数F_n满足：

$$F_0=0, F_1=1, \text{并且当 } n \geqslant 2 \text{ 时}, F_n=F_{n-1}+F_{n-2}。$$

上述数的序列称为斐波纳契序列，它在许多其他问题中出现。例如在10.4节，F_n作为高度为n的AVL树的结点数。我们这一节的目标是找出F_n的公式。

* * * 生成函数

我们使用生成函数(generating function)的方法，这种方法在许多其他应用中也很重要。生成函数是符号x的一个形式化的无限级数，它用斐波纳契数作为系数：

$$F(x)=F_0+F_1x+F_2x^2+\cdots+F_nx^n+\cdots$$

我们不关心这个级数是否收敛，或者x的值可能是什么，因为我们不会将x设置成任何特定的值。相反，我们只对生成函数执行形式化的代数操作。

接下来，我们乘上x的幂：

$$\begin{aligned} F(x)&= F_0+F_1x+F_2x^2+\cdots+F_nx^n+\cdots \\ xF(x)&= F_0x+F_1x^2+\cdots+F_{n-1}x^n+\cdots \\ x^2F(x)&= F_0x^2+\cdots+F_{n-2}x^n+\cdots \end{aligned}$$

然后从第一个等式减去后两个等式，得到：$(1-x-x^2)F(x)=F_0+(F_1-F_0)x=x$，由于$F_0=0$，$F_1=1$，并且当$n \geqslant 2$时，$F_n=F_{n-1}+F_{n-2}$，因此我们得到：$F(x)=\dfrac{x}{1-x-x^2}$。

* * * 封闭形式

$1-x-x^2$的根是$\dfrac{1}{2}(-1\pm\sqrt{5})$，采用部分分式的方法，我们能够重新安排$F(x)$的公式，得到：

$$F(x)=\frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi x}-\frac{1}{1-\psi x}\right)$$

其中$\phi=\dfrac{1}{2}(1+\sqrt{5})$且$\psi=1-\phi=\dfrac{1}{2}(1-\sqrt{5})$。

(通过将右部的两个分式放在公用分母上去检查$F(x)$的这个等式。)

* * * 解答

下一步是通过将右侧分式的分母变成1，从而展开右侧的分式：

$$F(x)=\frac{1}{\sqrt{5}}(1+\phi x+\phi^2x^2+\cdots-1-\psi x-\psi^2x^2-\cdots)$$

最后一步：$F(x)$的系数是斐波纳契数，因此使得这个等式两端的每个x的幂相等，我们得到：

$$F_n=\frac{1}{\sqrt{5}}(\phi^n-\psi^n)$$

ϕ和ψ的近似值是$\phi\approx 1.618034$和$\psi\approx -0.618034$。

* * * 黄金分割

斐波纳契数值的这个惊人的简单答案在几个方面都很有趣。首先，甚至不能马上看出为何右侧总是一个整数。其次，Ψ是一个负数，具有足够小的绝对值，我们通常将$F_n=\phi^n/\sqrt{5}$

舍入成最近的整数。最后，Φ 本身很有趣，它从古希腊时期就开始被研究，它通常称为黄金分割，而 Φ 与 1 的比率据说给出了最合意的矩形形状。帕台农神庙和许多其他的古希腊建筑的侧面就具有这一比率。

A.5 Catalan 数

这一节的目的是计算 n 个顶点的二叉树的数目，为了得到这个结果，我们需要绕点远路，顺着其他几个具有同样答案的问题去找出答案。我们称结果得到的数字为 Catalan 数，由于它们出现在许多明显无关的问题的答案中，从而变得相当有趣。

A.5.1 主要结果

定义 对 $n \geqslant 0$，第 n 个 Catalan 数定义为：

$$Cat(n)=\frac{C(2n,n)}{n+1}=\frac{(2n)!}{(n+1)!\ n!}$$

定理 A.7 当 $n \geqslant 0$ 时，具有 n 个顶点的不同二叉树的数目是第 n 个 Catalan 数 $Cat(n)$。

A.5.2 通过一一对应证明

1. 果园

首先，我们回忆定理 11.1 中具有 n 个顶点的二叉树与具有 n 个顶点的果园之间的一一对应，因此为了计算二叉树的数目，我们只需计算果园的数目。

2. 构成良好的括号序列

其次，我们考虑 n 个左括号'('和 n 个右括号')' 的所有构成良好的序列集。构成良好的序列意味着当从左向右扫描序列时，遇到的右括号的数目不会超过左括号的数目，因此'((()))'和'()()()'是构成良好的，但'())(()'不是，'(()'也不是，因为表达式中左括号的总数与右括号的总数并不相等。

引理 A.8 当 $n \geqslant 0$ 时，在 n 个顶点的果园与 n 个左括号和 n 个右括号构成良好的序列之间存在一一对应关系。

*** 括号表示形式

为了定义这种一一对应关系，我们首先回忆果园或者是空的，或者是有序树的一个有序序列。我们将果园的括号表示形式定义为它的树序列的括号表示形式，以与果园中的树同

样的次序将它们一个接一个地写下来。空果园的括号表示形式是空的。我们也记得有序树定义为由它的根顶点和子树的果园组成，因此我们将有序树的括号表示形式定义成由一个左括号'('后接根(的名字)，再接子树果园的括号表示形式，最后是一个右括号')'。

图 A.9 中显示了几个有序树和果园的括号表示形式。显然我们给出的相互递归的定义使任何果园产生惟一的括号表示形式，而且结果得到的括号序列是构成良好的。另一方面，如果我们从一个构成良好的括号序列开始，则最外面的括号对对应于果园中的树，而在这个括号对之内是关于树根及其子树果园的相应树的描述。这样，我们现在就得到了具有 n 个顶点的果园和具有 n 个左括号和 n 个右括号的构成良好的序列之间的一一对应关系。

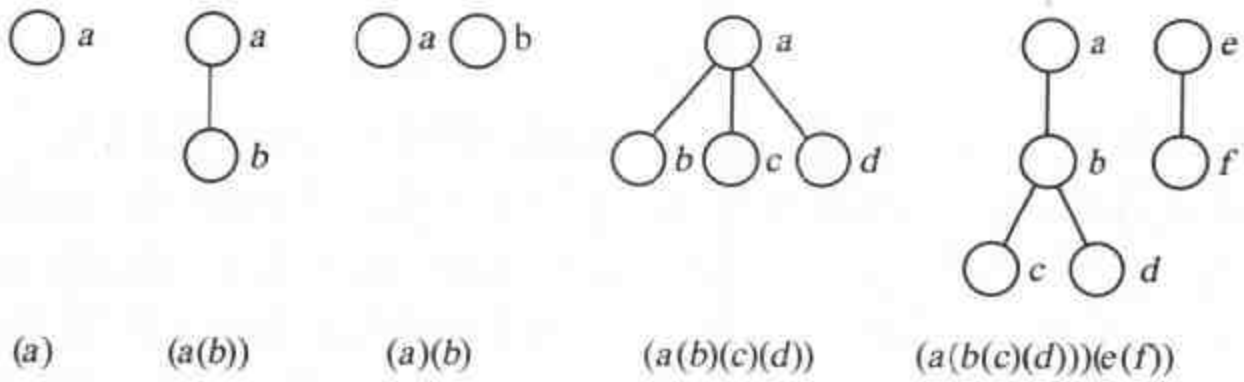


图 A.9　果园的括号表示形式

在计算果园的数目时，我们不关心加到顶点上的标记，因此我们将忽略标记，而采用我们列出的对应关系，现在将计算 n 个左括号和 n 个右括号的构成良好的序列的数目，在这些括号内没有其他东西。

3. 栈排列

我们指出，通过用+1 替换每个左括号，并用 -1 替换每个右括号，构成良好的括号序列就对应于+1 和 -1 的序列，使得从左边开始的部分和总是非负的，而它的总和是 0。如果我们将每个+1 看作将项压入到栈中，每个 -1 看作出栈，则部分和就是某给定时刻栈中的项数。从这一点可以看出，n 个对象的栈的排列数(见 2.1 节练习 E4)仍是由 Catalan 数给出答案的另一个问题。而且，如果我们从果园开始并执行一个完全的遍历(绕果园中的每个分支和顶点走，仿佛它是一面装饰的墙)，当每次向下走一个分支时计为 +1，而向上走一个分支时计为 -1(对每个叶子为+1 -1)，于是我们实质上再次得到与构成良好的序列的对应关系。

4. 任意的括号序列

我们最后一步是计算构成良好的括号序列的数目，但为了实现这个目的，我们改成计算所有非构成良好的括号序列的数目，并从所有可能的序列数中减去这个数。我们需要一个最终的一一对应关系：

引理 A.9　n 个左括号和 n 个右括号的非构成良好的序列恰好对应于 $n-1$ 个左括号和 $n+1$ 个右括号(以所有可能的次序)的所有序列。

为证明这个对应关系，我们从 $n-1$ 个左括号和 $n+1$ 个右括号的非构成良好的序列开始。k 是此序列中出现错误的第一个位置，因此位置 k 的元素是一个右括号，直到这个位置

时右括号比左括号多一个。因此严格在位置 k 右侧的右括号比左括号少一个，那么我们严格在位置 k 的右侧用右括号替换所有的左括号，并用左括号替换所有的右括号。结果得到的序列总共有 $n-1$ 个左括号和 $n+1$ 个右括号。

反之，我们从 $n-1$ 个左括号和 $n+1$ 个右括号组成的序列开始，假设 k 是右括号数超过左括号数的第一个位置(由于总共的右括号比左括号多，所以必然存在这样的位置)。我们再次在此序列的剩余部分中(在 k 之后的位置)用左括号替换右括号并用右括号替换左括号，因此我们得到一个有 n 个左括号和 n 个右括号的非构成良好的序列，这样我们已构造出想要的一一对应关系。

5. 证明结束

有了所有这些初步的对应关系，我们的计数问题简化成简单的组合问题。$n-1$ 个左括号和 $n+1$ 个右括号的序列数是从序列的 $2n$ 个位置中选择 $n-1$ 个由左括号占据的位置的方式数，也就是说，这个数是 $C(2n,n-1)$。由引理 A.9，这个数也是 n 个左括号和 n 个右括号的非构成良好的所有序列的数目。n 个左括号和 n 个右括号的所有序列数类似地是 $C(2n,n)$，因此构成良好的序列数是 $C(2n,n)-C(2n,n-1)$，它正好是第 n 个 Catalan 数。

因为所有的一一对应关系，我们还有：

推论 A.10 n 个左括号和 n 个右括号的构成良好的序列数、用栈可得到的 n 个对象的排列数、n 个顶点的果园数和 n 个顶点的二叉树数都等于第 n 个 Catalan 数 $Cat(n)$。

A.5.3 历史

令人吃惊的是，Catalan 数的发现并非因为前面的任何问题，而是由于几何问题。特别是 $Cat(n)$ 提供了将 $n+2$ 条边的凸多边形用 $n-1$ 条互不相交的对角线分成三角形的方式数。(见图 A.10。)这个问题好像是由 L. Euler 提出并由 J. A. v. Segner 在 1759 年解决的，它在

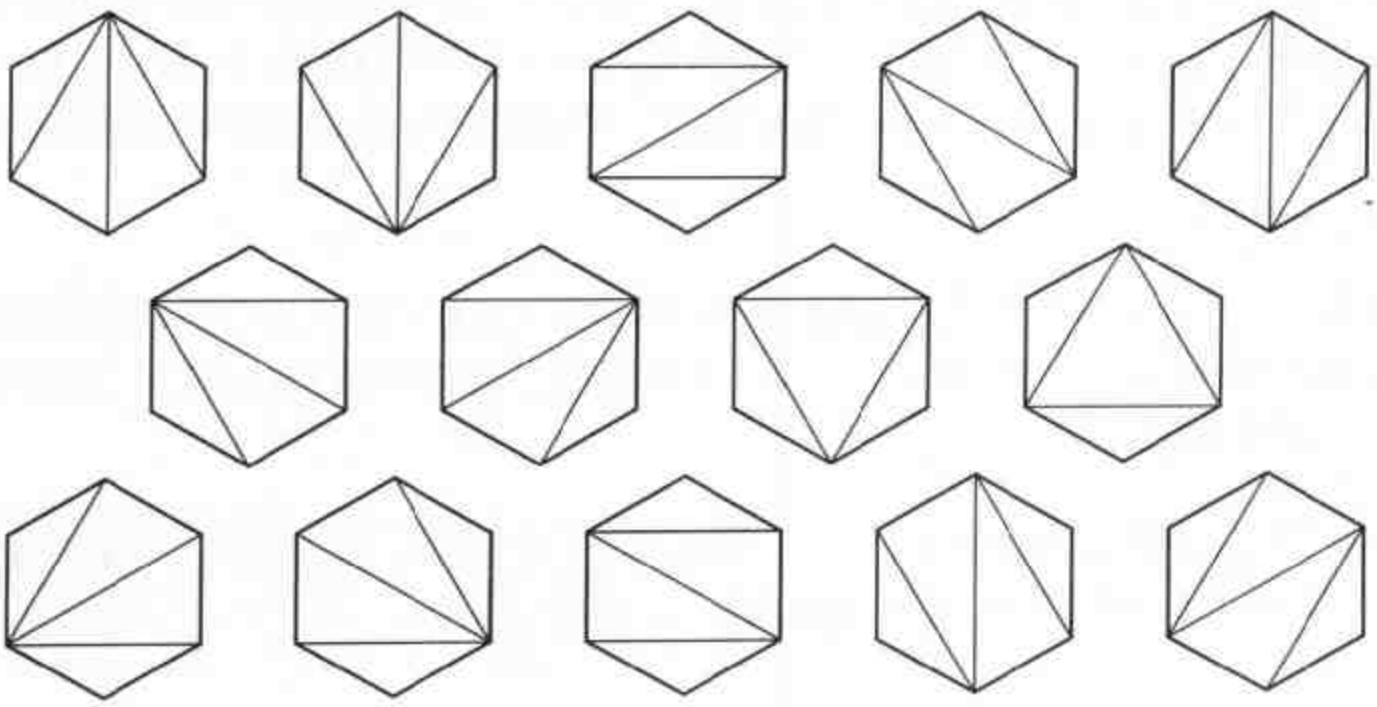


图 A.10 用对角线将六边形分成三角形

1838 年由 E. Catalan 再次解决。因此，有时结果数也称为 Segner 数，但更多的时候是称为 Catalan 数。

A. 5. 4　数学结果

我们用 Catalan 数大小的一些指示来结束这一节，图 A. 11 中给出了前 20 个值。

n	$Cat(n)$	n	$Cat(n)$
0	1	10	16,796
1	1	11	58,786
2	2	12	208,012
3	5	13	742,900
4	14	14	2,674,440
5	42	15	9,694,845
6	132	16	35,357,670
7	429	17	129,644,790
8	1,430	18	499,638,700
9	4,862	19	1,767,263,190

图 A. 11　前 20 个 Catalan 数

对较大的 n 值，我们能够使用 Stirling 近似去估计 Catalan 数的大小，当它将此近似用于 3 个分式中的每一个时，结果得到简化，我们得到：

$$Cat(n)\approx\frac{4^n}{(n+1)\sqrt{\pi n}}$$

当将它与图 A. 11 中的精确值进行比较时，此估计给出了 Stirling 近似精确值，例如，当 $n=10$ 时，Catalan 数的估计值是 17 007，而精确值为 16 796。

进阶参考书目

* * * 对数

对归纳证明、求和符号、整数幂的和及对数的更广泛的讨论出现于许多代数课本中，这些课本也提供了关于这些主题的例子和练习。关于对数的重要性和近似计算的精妙技术的讨论见 N. David Mermin，“Logarithms”，*American Mathematical Monthly* 87(1980)，1－7.

关于估计大数并用对数方法考虑它们的几个有趣例子的讨论见：Douglas R. Hofstadter，“Metamagical themas”，*Scientific American* 246，*no.* 5(*May* 1982)，20－34.

* * * 调和数

下面这篇非技术性的文章给出了关于调和数的几个令人惊奇和有趣的应用：Ralph Boas，“Snowfalls and elephants，pop bottles and π”，*Two-Year College Mathematics Journal* 11(1980)，82－89.

关于调和数和阶乘的详细的估计(Stirling 近似)是引自 Knuth 第 1 卷的第 108～111 页，在这里可以找到详细的证明。Knuth 第 1 卷也是有关排列、组合和相关主题的更多信息的极好来源。

＊＊＊阶乘

Stirling 近似的原始引用是：James Stirling，*Methodus Differentialis*（1730），p. 137.

＊＊＊组合数学

与各种集合或对象类的枚举相关的数学分支称为组合数学，这门计数科学能在非常简单的层次引入，或者以较高的复杂性进行研究。有两本初级课本包含了这里引入的许多思想的进一步展开，它们是：Gerald Berman and K. D. Fryer，*Introduction to Combinatorics*，Academic Press，1972. Alan Tucker，*Applied Combbinatorics*，John Wiley. New York，1980.

＊＊＊斐波纳契数

斐波纳契数的出处几乎会在所有关于组合数学的书籍中出现，在 Knuth 第 1 卷第 78～86页亦是如此，书中包含一些有趣的历史和许多有关的练习。实际上在下面这本书中有斐波纳契数的说明：Peter Stevens，*Patterns in Nature*，Little，Brown，Boston，1974.

人们已经并且还将继续发现斐波纳契数的几百种特性，这些特性经常发布在研究期刊 *Fibonacci Quarterly* 中。

＊＊＊Catalan 数

Catalan 数的出处（用于凸多边形的三角形分割）出现在被引用的有关组合数学书的开始（Berman and Fryer，pp. 230－232）。Knuth 第 1 卷第 385～406 页枚举了几类树，包括用于二叉树的 Catalan 数。提供了关于 Catalan 数的历史和应用的 46 种参考书目的列表见：W. G. Brown，“Historical note on a recurrent combinatorial problem”，*American Mathematical Monthly* 72（1965），973－977.

下面的文章说明了 Catalan 数的许多其他应用：Martin Gardner，“Mathematical games”（regular column），*Scientific American* 234，no. 6（June，1976），120－125.

关于 Catalan 数的出处的原始引用是：J. A. v. Segner，“Enumeratio modorum，quibus figuræ planæ rectilinæ per diagonales dividuntur in triangula”，*Novi Commentarii Academiæ Scientiarum Imperialis Petropolitanæ* 7（1758－1759），203－209. E. Catalan，“Solution nouvelle de cette question：un polygone étant donné，de combien de manieres peut-on le partager en triangles au moyen de diagonales?”，*Journal de Mathématiques Pures et Appliquées* 4（1839），91－94.

附录B　随　机　数

随机数是让计算机程序产生许多结果的一个有用工具。这个附录简单地处理随机数的产生，区别几种不同的随机数产生方法，然后将它们作为C++类Random的方法实现。

B.1　介绍

多样性是生活的情趣。另一方面，计算机往往是完全可以预测的，因此显得相当呆滞。随机数提供了一种给计算机程序注入不可预测性的方法，因而，有时使它们能更好地模拟外部事件。当随机数作为计算机游戏、图形显示或仿真的一部分时，它能增加许多乐趣，而且当程序重复运行时，它能在一定范围内表现与其模拟的自然系统互不相像的行为。

*** 系统随机数生成器

头文件<cstdlib>和<stdlib.h>提供了C++系统中的随机数生成例程，这些例程能用于代替此处开发的程序。但有时系统随机数生成器不是很好，因此值得去了解如何构造一个更好的随机数生成器。

在任何情况下，我们应把这一节中的随机数生成器和所有的子程序看作是产生随机数的软件包。一旦我们开发出这个软件包，则应能将它用于仿真程序或我们希望的任何其他应用中，但不必洞察它的内部是如何运行的。因此这一节的所有细节(相当偏向数学)应看作是实现的一部分，在应用中使用随机数时不必关心此实现。

我们将设计一个Random类来实现这些目标，它的方法将产生和返回各种随机数。

B.2　策略

*** 伪随机数的种子

我们用于生成随机数的思想是从一个数开始，然后运用一系列的算术运算，它们将产生与第一个数没有明显联系的另一个数。因此，我们产生的数并不真正是完全随机的，每个数都在一定程度上依赖于它的前趋，所以更合适地讲它们应该是伪随机数。我们使用的(和同时改变的)数字称为种子(seed)。

*** 不可再生的行为

如果每次程序运行时种子都从同一个值开始，则整个伪随机数序列将正好一样，而且我们能再次生成我们得到的任何实验结果。然而，万一客户程序希望使用的随机数不仅是不可预测，而且也是不可再生的，则我们应当包含一种选择，按以秒度量的精确时间来初始化这个种子。因为这个时间在每次程序运行时最可能不同，因此每次客户程序运行时，此初始化操作将导致不同的行为。例如，这样的不可再生行为适合于实现计算机游戏。

*** 参数:pscudo

我们通过为Random类创建一个构造函数来实现两种可能的初始化操作，这个构造函数将使用参数bool pseudo。当pseudo的值是true时，将由一个预定义的种子产生随机数，

而当 pseudo 是 false 时，将产生不可再生的随机数。这样，每次客户程序一开始就可自动地执行期望的初始化，同时客户程序不需要特殊的努力去初始化种子。

* * * 种子私有

种子由随机数生成器使用和改变，但即使可能，它也不应被应用程序使用甚至访问。也就是说，随机数包的用户不需要意识到种子变量的存在。因此，种子不应该作为随机数生成器的一个参数，而且由用户在用户程序中把它定义为全局变量而更少地初始化亦是不合理的。

因此我们将种子定义为 Random 类的私有数据成员。这样，种子对类中的所有方法和辅助函数都是可以访问的，但从类外面却根本不能访问到种子。

现在我们可以确定 Random 类的框架：

```
class Random {
public:
    Random(bool pseudo=true);
//   Declare random-number generation methods here.
private:
    int reseed();                  //  Re-randomize the seed.
    int seed,
        multiplier,add_on;         //  constants for use in arithmetic operations
};
```

B.3 程序设计

Random 类的核心是用于更新种子的辅助函数 reseed，它被调用以提供所有其他方法的随机行为。此函数如下操作：

```
int Random::reseed( )
/* Post: The seed is replaced by a psuedorandom successor. */
{
  seed=seed * multiplier +add_on;
  return seed;
}
```

在这个函数中，我们执行一个乘法和一个加法，然后丢掉结果的最高有效部分，仅保留较低有效的却更随机的数字。

常量 multiplier 和 add_on 被存储为 Random 对象的数据成员。它们不应被随机选择，而应该仔细选择以确信结果通过了各种随机测试。例如，由下面的 Random 对象的构造函数赋的值看起来在 16 位计算机上运行得相当好，但对其他的机器则应该做出其他的选择。

```
Random::Random(bool pseudo)
/* Post:  The values of seed, add_on, and multiplier are initialized. The seed is
          initialized randomly only if pseudo==false. */
{
  if (pseudo) seed=1;
  else seed=time(NULL)%max_int;
  multiplier=2743;
  add_on=5923;
}
```

用于产生不可预测的种子的函数 time()来自头文件 time. h，它测量从 1970 年开始以来所逝去的时间的秒数。

1. 实值

我们不允许客户代码直接使用函数 reseed，而是将它的结果直接转换成更直接有用的 3 种形式之一。

* * * 均衡分布

第一种形式模拟了大多数计算机系统的随机数生成器，它作为结果返回一个均衡分布在 0 和 1 之间的实数。均衡分布意味着：如果在此方法的域内取两个同样长度的区间，则结果落入一个区间和在落入另一个区间具有相等的可能性。因为我们方法的域是从 0 到 1，因此这个定义意味着结果在任何子区间的概率应等于那个子区间的长度。

在结果域上还有一个限制：0 可以作为结果出现而 1 不可以。按数学术语来说，这个域是半开区间[0,1)。

以下是得到的 Random 类的方法。它只是将来自 reseed 的结果转换到希望的域。为了执行这个转换，首先必须设置 max_int 的值去记录能存储的最大整型值，这个值可以从头文件<limits>，<climits>或<limits. h>中的某一个获得。

```
double Random :: random_real()
/*Post: A random real number between 0 and 1 is returned. */
{
  double max= max_int +1. 0;
  double temp= reseed();
  if (temp<0) temp= temp +max;
  return temp/max;
}
```

2. 整数值

随机数的第二种通用形式是整数。然而，我们不能有意义地谈及随机整数，因为整数的个数是无限的，但可以在计算机上表达的整数个数是有限的，因而真正的随机整数是计算机上可以表达的整数的概率是 0。我们改成仅考虑 low 和 high 这两个整数之间(包括 low 和 high)的域内的整数。为了计算这样的一个整数，我们从[0,1)之间的一个随机实数开始，将它乘以期望域中的整数个数 high - low + 1，并将结果截断为一个整数，然后再加上 low 以将结果放到期望的整数域中。我们也将此操作作为 Random 类的方法实现。

```
int Random :: random_integer(int low, int high)
/*Post: A random integer between low and high (inclusive) is returned. */
{
  if (low > high) return random_integer(high, low);
  else return ((int) ((high- low +1) *random_real( ))) +low;
}
```

3. 泊松值

* * * 期望值

随机整数第三种更复杂的形式为 3. 5 节的飞机场模拟所需要，它称为随机整数的泊松

分布。我们从称为随机数的期望值 v 的一个正实数开始，如果一个非负整数的序列满足期望值 v 的泊松分布，则在一个较长的子序列上，这些整数的平均值接近于 v。

＊＊＊例子

例如，如果从期望值 1.5 开始，则我们可能有一个序列 1,0,2,2,1,1,3,0,1,2,0,0,2,1,3,4,2,3,1,1,...，如果计算这个序列的子序列的平均值，可以发现平均值有时小于 1.5，有时大于 1.5，但慢慢地，它变得更可能接近于 1.5。

下面的函数根据泊松分布生成伪随机数，有关此方法的推导和正确性证明需要微积分和高等数学的统计知识，这远远超出了本书的范围，但这并不意味我们不能运用这个理论去计算我们想要的数。这个结果是 Random 类的第三种方法：

```
int Random::poisson(double mean)
/*Post: A random integer, reflecting a Poisson distribution with parameter mean, is
        returned.*/
{
  double limit=exp(-mean);
  double product=random_real();
  int count=0;
  while (product > limit){
    count++;
    product *= random_real( );
  }
  return count;
}
```

这里的函数 exp(**double** x)是指数函数，它的实现来自于<math.h>或<cmath>库之一。

程序设计项目 B.3

P1. 编写一个驱动程序，用于测试本附录设计的 3 个随机数函数。对每一个函数，计算它对一系列调用(调用的次数由用户说明)返回的平均值。对 random_real，平均值应大约是 0.5；对 random_integer(low,high)，平均值应大约是(low+high) / 2，这里 low 和 high 由用户给定；对 poisson(expected_value)，平均值应近似于用户描述的期望值。

P2. 用于随机整数均衡分布的一个测试是为了了解所有的概率是否经常近似同等地出现。建立一个整数计数器数组，从用户处得到要使用的位置数和待试验的次数，然后在指定的域内重复产生一个随机整数并递增合适的数组单元。最后，存储在所有单元中的值将近似是同样的。

P3. 推广前一项目中的测试，使用一个矩形数组和两个随机整数去确定递增哪个单元。

＊＊＊剪刀-布-石头

P4. 在某个孩子游戏中，两个游戏者同时伸出一只手，以某种方式表示剪刀、布或石头中的一种。规则是剪刀打败布(因为剪刀剪布)，布打败石头(因为布包住石头)，石头打败剪刀(因为石头砸坏剪刀)。编写一个程序，模拟与人玩这个游戏，人可以在每轮键入 S、P 或 R。

＊＊＊Hamurabi

P5. 在 Hamurabi 游戏中，你是一个小国的国王。开始时你拥有 100 个人，100 蒲式耳谷物和

100 亩田地。你必须决定在每年分配这些资源，你可以让人们多吃你的谷物，可以多种植植物，或者可以买卖土地交换谷物。土地的价格（以每亩多少蒲式耳谷物计算）将随机决定（在 6 和 13 之间）。老鼠会消耗掉某个随机百分比的谷物，且在一年之中会随机发生其他的事件，这些事件包括：

- 瘟疫
- 丰收
- 人口激增
- 水灾
- 农作物病虫害
- 邻国破坏！每亩土地只卖两蒲式耳谷物
- 卖方市场

在每年年终，如果有太多的（这也是随机的）人由于缺少谷物挨饿，他们将做出反抗并推翻你的统治。编写一个程序，确定随机事件并在你每年进行决策时跟踪你的王国的发展。

＊＊＊随机步行

P6. 如图 B.1 所示，一个醉汉离开酒吧后，试图走路回家。酒吧和家之间的街道形成了一个矩形网格。每次当醉汉到达一个拐角时，他随机决定下一步要走的方向。而且，他从未在网格之外徘徊。

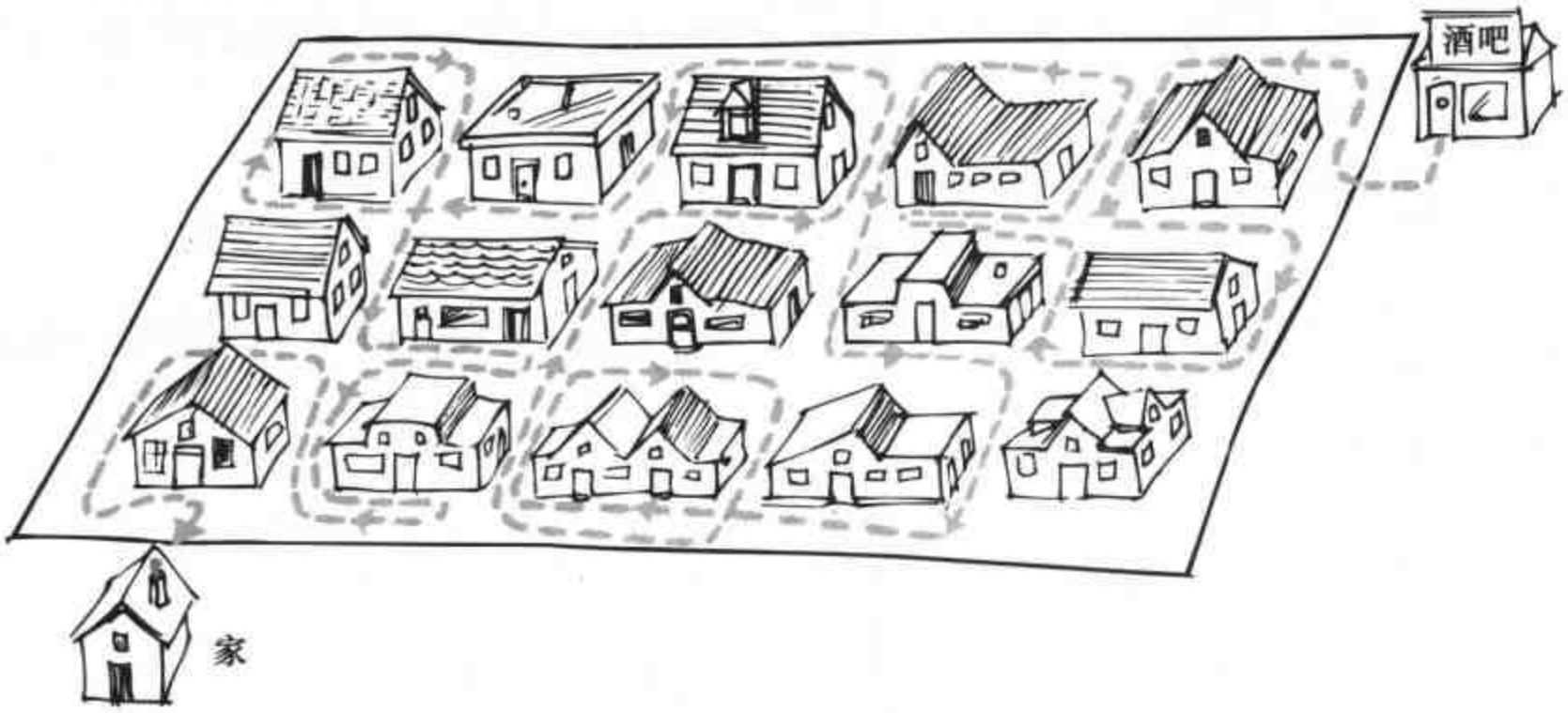


图 B.1　随机步行

(1) 编写一个程序模拟随机步行，要求方格中行和列的数目是可变的。你的程序应计算醉汉在同一方格上步行许多随机步骤后，平均花了多长时间回到家。研究这个数如何依赖于方格的形状和大小。

(2) 为了改善他的运气，让醉汉走到离酒吧更近的地方——到方格左上角上的一个房间。修改此模拟程序，看看他现在能以多快的速度到家。

(3) 修改最初的模拟程序，使得如果醉汉碰巧返回到酒吧，则他进入酒吧且结束步行。找出（依赖于方格的大小和形状）醉汉成功到家的次数百分比是多少。

(4)修改最初的模拟程序,以如下方式为醉汉提供一些记忆力以帮助他。每次当他到达一个角落时,如果他在当前步行之前曾到过那儿,则他记得已经走过了什么街道,于是会选择一条新的街道。如果他已经试过了那个角落的所有街道,则会随机决定现在要走哪一条。他多快能回到家?

* * * 查找和排序数据的生成

P7. 编写一个程序生成整数文件,整数的形式要适合测试和比较各种查找、排序和信息检索程序。测试和比较这些程序的一种合适的办法是用此程序创建一小组大小变化且次序多样的文件,然后用 CPU 计时单元比较在同样数据文件上运行的各种程序。

程序应产生直到 15 000 个整数的任意规模的文件,整数以下列几种方法之一排列,这些方法包括:

(1)随机次序,允许重复键

(2)随机次序,不允许重复键

(3)增序,允许重复

(4)增序,不允许重复

(5)降序,允许重复

(6)降序,不允许重复

(7)接近但不完全有序,允许重复

(8)接近但不完全有序,不允许重复

接近有序的文件能通过输入用户指定的数据修改。程序应使用伪随机数去生成无序文件。

进阶参考书目

Knuth,第二卷,第 1～177 页,提供了伪随机数生成器、测试方法和相关主题的一个最综合的论述。

附录C　软件包和实用函数

这个附录描述了我们在整本书中已经使用过的实用软件包中的类、定义和函数，它也列出了在本书各个部分中设计和使用的其他软件包的内容，它以一个程序执行的计时软件包作为本附录的结束。

C.1　软件包和C++转换单元

*** 软件包的目的

大型C++程序通常要分成几类不同的文件，程序分割成文件的方式比较重要，因为它有助于人类阅读者理解大型项目的逻辑上相互独立的部分。这种分割方式甚至能帮助编译器维护项目的逻辑结构。而且就如我们所见，程序的关键构造块能重复地用于其他程序当中。这样，通过明智地分割程序，我们可以创建能插入其他项目的软件包，从而便于代码重用。例如，贯穿本书，只要方便，我们就能调用2.2节设计的Stack方法的软件包。

软件包通常，但不总是由密切相关的任务集合组成。例如在本书中，我们使用一个软件包计算和了解用于一个程序的CPU时间，而使用另一个软件包(在附录B中设计)计算伪随机数。我们的实用工具包收集了一些经常使用的函数，即使它们之间根本互不相关。

*** 数据结构

软件包特别适合存放数据结构。因此，例如要求Stack栈结构的客户程序仅需要调用Stack软件包以得到此数据结构。当然，客户程序需要知道软件包所提供的方法，但正如我们所知，客户程序不必且实际上不应关心软件包的实现细节。

*** 接口和实现

通常，软件包应提供包含客户程序需要知道的细节的接口和可能对客户程序隐藏的实现。例如2.2.2节说明了栈操作但根本没有涉及实现。任何栈软件包的接口部分将包含完全同样的说明，实现部分将包含接口中指定的每个栈方法的完整细节。

*** 实现的改变

由于接口部分保持完全一致，因此使用栈软件包的程序不需要知道将使用哪种实现。事实上，如果对实现进行了正确的程序设计，程序不可能区分出软件包中是哪种实现(或许除了通过测量所完成的各种操作的速度)。

*** 软件包的文件

虽然我们在构建程序时经常不会考虑可重用的软件包，但在开始实现程序时，我们必须使用C++编译器能接受的单元，即使用文件。当我们考虑编译器完成的工作和链接器的后续操作时，我们应看到，在实现中每个软件包通常被分割成一对文件。这些文件非常自然地对应于我们挑选出来作为软件包接口和实现的概念。

*** 声明或头文件

- 第一个文件由软件包的元素声明组成，包括软件包中的类、函数、对象和模板的名字

和类型信息，它称为软件包的声明文件或头文件。声明文件通常带有扩展名.h，用于函数的声明信息称为函数原型。

* * * 定义或代码文件

• 第二个文件给出了软件包中所有元素的完整定义，包括它们的实现代码。这个文件称为软件包的定义文件或代码文件。此文件通常带有扩展名.c（或在其他系统中为.C，.cpp，.cxx 或.cc 之一）。

* * * 转换单元

* * * 对象文件

编译器在任何时候只能处理单个程序文件。这个文件可能使用 # include 指令读入其他文件或者使用其他指令选择要编译的代码部分。在这样的情况中，编译器会在执行任何编译之前跟随这些指令。跟随所有 # include 和其他指令所产生的结果代码称为转换单元。编译器在转换单元上操作并将它转变为目标文件。编译器能访问转换单元中所有元素的定义这一点很重要。特别地，我们必须用 # include 包含与转换单元中调用的任何软件包对应的.h 文件。

* * * 链接

组成程序的不同目标文件被链接器连到一起形成可执行代码。链接器必须确信程序的每个元素都得到完全的定义，而且在所有的不同转换单元之间，只有一个这样的定义。这一点意味着.c 文件应该只包含在程序的一个转换单元中。通常我们通过编译.c 文件本身来保证这一点，而且我们决不会在另一个程序文件中包含.c 文件。

* * * 信息隐藏

软件包分成独立的.h 文件和.c 文件除了有助于大型项目的编译和链接外，还有效地维持了信息隐藏的原理。.h 文件提供了软件包的接口，.c 文件给出了实现。

C.2 课文中的软件包

本书中开发的大多数软件包是针对所学的各种数据结构，这些软件包（经常是作为练习和项目）是在本书的各部分中介绍数据结构时开发的。此处是我们开发的大多数软件包的一个列表，同时给出了它们在本书中的章节号。对表达我们基本数据结构的软件包，我们列出它们的方法，然后给出提供方法实现的书中章节号。最后两个软件包提供了通用目的实用程序代码和程序计时方法，已用于整本书中，但我们随后将在此附录中予以讨论。

Stack package:

```
Stack();
bool empty() const;
Error_code pop();
Error_code top(Stack_entry &item) const;
Error_code push(const Stack_entry &item);
```

Contiguous stack（顺序栈） 2.2 节
Linked stack（链栈） 4.2 节

Queue package:

```
Queue();
bool empty() const;
Error_code serve();
```

```
    Error_code append(const Queue_entry &item);
Error_code retrieve(Queue_entry &item) const;
```

Contiguous queue(顺序队列)	3.2节
Linked queue(链式队列)	4.4节

List package:

```
    List();
    int size() const;
    bool full() const;
    bool empty() const;
    void clear();
    void traverse(void(*visit)(List_entry &));
    Error_code retrieve(int position, List_entry &x) const;
    Error_code replace(int position, const List_entry &x);
    Error_code remove(int position, List_entry &x);
    Error_code insert(int position, const List_entry &x);
```

Contiguous list(顺序表)	6.2.2节
Linked list(链表)	6.2.3节
Doubly linked list(no head)(无头双向链表)	6.2.5节
Simply linked list in array(数组中的简单链表)	6.5节
Doubly linked list in array(数组中的双向链表)	6.5节练习E5

Tree packages:

```
    Binary_tree();
    bool empty() const;
    void preorder(void(*visit)(Entry &));
    void inorder(void(*visit)(Entry &));
    void postorder(void(*visit)(Entry &));
    int size() const;
    void clear();
    int height() const;
    void insert(const Entry &);
```

Binary tree	10.1节

```
    Error_code remove(const Record &);
    Error_code tree_search(Record &) const;
```

Binary search tree(二叉查找树)	10.2节
AVL tree(AVL树)	10.4节
Red-black tree(红-黑树)	11.4节
Splay tree(伸展树)	10.5节

B-Tree package:

```
    B_tree();
    Error_code search_tree(Record &target);
    Error_code insert(const Record &new_node);
    Error_code remove(const Record &target);
    void print();
```

B-tree(B-树)	11.3节
Deque(双端队列)	3.3节练习E10
Character string(字符串)	6.3节
Sortable list(排序表)	第8章
Trie	11.2节
Polynomials(多项式)	4.5节
Look-ahead for games(预测游戏)	5.4节
Random numbers(随机数)	附录B
Utility package(实用程序软件包)	C.3节
Program-timer package(timer软件包)	C.4节

C.3 实用程序软件包

一些程序语句，虽然逻辑上互不相关，但使用却非常频繁，因此可以将它们集中到一起形成一个实用程序软件包，使得所有程序都可利用它。因此本书开发的几乎所有程序都包含语句#include "utility.h";，它允许程序访问这个实用程序软件包的内容。

* * * 系统依赖性

实用程序软件包的一个目的是为广泛使用的任务如输入和输出包含系统库，这些库文件的名字因使用的是ANSI版还是老版本的C++而有所不同。通过将这些依赖系统的特征收集在单一软件包中，可以确信其他的程序不依赖于系统中C++的精确版本。这一措施大大改善了程序的可移植性，即，我们可以在不同的系统中用不同的编译器轻松地编译它们。

ANSI版的接口文件utility.h如下：

```
using namespace std;
#include <iostream>          //  standard iostream operations
#include <limits>            //  numeric limits
#include <cmath>             //  mathematical functions
#include <cstdlib>           //  C string functions
#include <cstddef>           //  C library language support
#include <fstream>           //  file input and output
#include <cctype>            //  Character classification
#include <ctime>             //  date and time functions
bool user_says_yes();
enum Error_code{success,fail,range_error,underflow,overflow,fatal,
                not_present,duplicate_error,entry_inserted,entry_found,
                internal_error};
```

* * * * 标准库

在ANSI C++中，标准库包含在**namespace** std中，因为这个原因，为了能不带域解析运算符而使用标准库，我们在工具包的开头放入ANSI C++指令**using namespace** std;。

软件包的下一部分由包含各种系统库的指令组成。我们附加在#include命令后的注释描述了该包含文件的目的。最后，我们声明将在许多程序中使用的函数uscr_says_yes和Error_code的类型。

* * * C++旧版

用于相当老的C++实现的公用程序接口文件的版本可能需要包含布尔类型的仿真，还需要包含一个稍微不同的标准头文件。例如，下面的接口可能适合带有老版本C++编译器的某些系统：

```
#include <iostream.h>        //  standard iostream operations
#include <limits.h>          //  numeric limits
#include <math.h>            //  mathematical functions
#include <stdlib.h>          //  C-string functions
#include <stddef.h>          //  C library language support
#include <fstream.h>         //  file input and output
#include <ctype.h>           //  Character classification
#include <time.h>            //  date and time functions
```

```
typedef int bool;
const bool false=0;
const bool true=1;
bool user_says_yes();

enum Error_code{success,fail,range_error,underflow,overflow,fatal,
                not_present,duplicate_error,entry_inserted,entry_found,
                internal_error};
```

* * * user_says_yes

实用程序软件包的实现文件中所需要的惟一代码是函数 user_says_yes()的定义，因为这是标准库中惟一没有发现的实用函数。下面的实现适用于所有的 C++编译器：

```
bool user_says_yes()
{
  int c;
  bool initial_response= true;
  do{                                  //  loop until an appropriate input is received.
    if(initial_response)
       cout<<"(y,n)?"<< flush;
    else
      cout<<"Respond with either y or n:"<< flush;
    do{                                //  ignore white space.
      c= cin.get();
    }while(c=='\n'||c==' '||c=='\t');
    initial_response=false;
  } while(c!='y'&&c!='Y'&&c!='n'&&c!= 'N');
  return(c=='y'||c=='Y');
}
```

C.4　计时方法

在比较不同的算法和数据结构时，了解一个程序或程序段在与另一个比较时所花费的计算机时间通常是有用的。因此为了这个目的，我们开发了一个实现 Timer 类的软件包。构造函数 Timer 启动定时器工作，而如果我们希望重新设置定时器，则此软件包提供了一个方法 reset()。方法 elapsed_time 返回从 Timer 对象启动或最后调用 reset 后所使用的 CPU 时间，因此 elapsed_time 非常类似于秒表上显示的一次计时。

C++系统提供了头文件<ctime>或<time.h>，它包含函数 clock()给出 clock_t 类型的时间值。通过将时间间隔除以一秒内时钟的嘀嗒数 CLK_TCK，我们能计算逝去的时间。

文件 timer.h 包含了 Timer 类的下列定义。

```
class Timer {
public:
  Timer();
  double elapsed_time();
  void reset();
private:
  clock_t start_time;
};
```

timer. c 中包含方法的编码：

```
Timer::Timer()
{
  start_time=clock();
}
double Timer::elapsed_time()
{
  clock_t end_time=clock();
  return((double)(end_time- start_time))/((double)CLK_TCK);
}
void Timer::reset()
{
  star_time=clock();
}
```

附录 D　程序设计规则、启示和易犯的错误

这个附录收集了本书所有章中出现的程序设计规则、启示和易犯的错误，并将它们按照主题进行排列，从课本中研究的数据结构和算法的全面综述开始，然后是设计数据结构和算法的通用准则，之后是递归，最后是计算机程序的构造、测试和维护。

D.1　数据结构和算法的选择

D.1.1　栈

1. 栈是最简单的数据结构之一，需要时使用栈。

2. 在任何需要反转数据的问题里，考虑使用栈保存数据。

D.1.2　表

3. 不要混淆顺序表和数组。

4. 当采用通用表时，首先明确地决定需要什么操作，然后选择能最简单地完成那些操作的实现。

＊＊＊链式和顺序

5. 在表的链式和顺序实现之间做出选择时，考虑表上必要的操作。链表对插入、删除和重排更加灵活，而顺序表允许随机访问。

6. 当表中的项较小且算法简单时，顺序表一般需要较少的计算机内存、计算机时间和程序设计工作。当表中存放着较大的数据元素时，链表一般更节省空间和时间，并经常也节省程序设计工作。

7. 动态内存和指针允许程序动态地适应广泛的应用规模，且提供了不同数据结构间内存分配的灵活性。静态内存(数组和下标)有时在事先能完全确定大小的应用中更有效。

D.1.3　查找方法

8. 顺序查找虽然较慢但却很健壮，在短表或对未知表中的键是否合适排序的情况下使用顺序查找。

9. 如果必须对二分查找重新进行程序设计，务必特别小心，验证你的算法正确，并在所有极端情况下测试它。

D.1.4　排序方法

10. 许多计算机系统有通用目的排序例程。如果你能访问这些例程，并且它足够于你的应

用，那么就使用它们，而不是从头开始编写一个排序程序。

11. 在选择排序方法时，考虑键在排序前通常的排列方式、应用的规模、程序设计可用的时间、节省计算机时间和空间的需要、数据结构实现的方式、移动数据的代价和比较键的代价。

12. 归并排序、快速排序和堆排序是强大的排序方法，它们比简单方法更难以编程，但它们对大型的表效率更高。仔细考虑应用，确定为实现这些复杂算法之一而需要付出的额外工作是否合理。

13. 堆排序是一种比较保险的方法：它通常比快速排序慢，但它保证能在 $O(n \log n)$ 次键比较内完成排序，而这一点连快速排序也不是总能做到。

* * * 优先队列

14. 优先队列对许多应用来说很重要，而堆提供了优先队列的一种极好的实现。

D.1.5 表格

15. 使用数据的逻辑结构决定要使用哪一种表格：普通数组、特殊形状的表格、反转表格系统或哈希表。选择允许所要求操作并满足问题的空间要求的最简单的结构。不要编写复杂的函数去节省那些保留不用的空间。

* * * 下标函数、访问数组

16. 让数据的结构帮助你确定访问数据表格时是下标函数还是访问数组更好。只要可能，就使用程序设计语言内含的特征。

* * * 哈希表

17. 在使用哈希表时，让数据的性质和必需的操作帮助你在链接和开放定址之间做出决策。链接在要求删除操作、记录相对较大或面临溢出问题时一般更可取；开放定址在单个记录较小及没有溢出哈希表的危险时一般更可取。

18. 哈希函数通常需要根据访问此哈希表的键的种类进行自定义。在设计哈希函数时，在同时保持哈希表上相对均匀的键分布时，尽可能保持简单且少量的计算。在计算中不强制使用键的每个部分。对重要的应用，应当使用计算机实验哈希函数的几个变种，并寻找快速的计算和键的平均分布。

D.1.6 二叉树

19. 将二叉查找树看作是有序表的一种选择（实际上是作为实现抽象数据类型 list 的一种方式）。二叉查找树以每个结点中的额外指针成员为代价，在保持链表插入、删除和重排的灵活性的同时，允许随机访问树中所有的结点（$O(\log n)$ 次键的比较）。

20. 将二叉查找树看作是表格的一种选择（实际上是作为实现抽象数据类型 table 的一种方式）。二叉查找树以 $O(\log n)$ 而不是 $O(1)$ 的访问时间为代价，在保持表格提供的随机访问的优点的同时，允许以键指定的次序遍历数据结构。

21. 在选择数据结构时，总是仔细考虑所需要的操作。当随机访问、以预定的次序遍历、插入和删除的灵活性等要求都需要满足时，二叉树尤其合适。

＊＊＊不平衡查找树

22. 当选择数据结构和算法时，警惕高度不平衡二叉查找树的可能性。如果进来的数据可能是以随机次序，则普通的二叉树证明完全足够使用。如果数据可能以有序或接近有序的次序进来，则算法应采用适宜的操作。如果只存在一点严重不平衡的可能性，则可以忽略。而如果在一个大的项目中存在更大可能的严重不平衡，则软件中要在合适的位置检查树的平衡性，必要时要重建树的平衡性。对于所有时间内都必须保持对数访问时间的应用，AVL 树以较小的计算机时间和空间代价提供了接近完美的平衡，但它却要求付出可观的程序设计代价。如果树必须动态适应数据频繁的改变，则伸展树可能是最好的选择。

＊＊＊递归结构

23. 二叉树采用递归定义，因此操作二叉树的算法一般最好采用递归编写。在用二叉树进行程序设计时，要意识到问题通常与递归算法有关。确信算法在任何条件下都会终止，而且算法正确地处理空树的简单情况。

24. 虽然二叉树通常以链式结构实现，但也要意识到仍存在其他实现的可能性。在用链式二叉树进行程序设计时，谨记用链表进行程序设计时易犯的错误。

D.1.7　通用树

25. 对于问题建模和数据组织，树都是灵活而功能强大的结构。在问题求解和算法设计中使用树时，在考虑实现细节前，首先确定所需要的树的种类(有序树、有根树、自由树或二叉树)。

26. 大多数树能通过递归进行描述，因此它们关联的算法通常最好用递归方式设计。

＊＊＊树结构的选择

27. 对信息检索问题，在选择将使用的数据结构时，要考虑规模、数目、记录的位置及元素的类型和结构。对于小型记录或较少数目的元素，要使用高速的内存，二叉查找树可能证明是足够用的。对磁盘文件的信息检索，运用多路分支的方法如 trie、B-树和哈希表通常是最优的。trie 特别适合用于键的结构是一个符号序列和键集在所有可能的键的集合中相对密集这样的应用。对于其他的应用，把键看作单个单元的方法通常证明是出众的。B-树及其各种推广和扩展，能有效地应用于许多关于外部信息检索的问题中。

D.1.8　图

28. 图提供了描述许多应用的本质特征的极好方法，因而便于说明基本问题和明确表达求解问题的算法。图有时以数据结构出现，但更多的时候是以有助于问题求解的数学抽象出现。

29. 可以使用不同种类的数据结构以许多方法实现图。延缓图的实现选择，直至问题求解和算法设计阶段图的应用得到了充分的理解。

＊＊＊图的遍历

30. 许多应用都需要图的遍历。让应用决定遍历的方法：深度优先、广度优先或某种其他次

序。深度优先遍历自然是递归的(或能够使用栈)。广度优先遍历通常使用队列。

31. 贪心算法仅仅表示在设计图算法中有用的许多范例的一个样本。更多的方法和例子,查阅参考书目。

D.2 递归

32. 在算法的初始设计中应自由地使用递归。当求解的主要步骤是由将问题简化的一个或几个更小的问题组成时,递归方法特别合适。

*** 从例子设计

33. 研究几个简单的例子以了解是否应该使用递归以及递归将如何工作。
34. 尝试用公式表示更普遍适用的方法。询问:"如何能将这个问题分成几个部分?"或者"如何完成中间的关键步骤?"
35. 询问问题的剩余部分能否以同样的或类似的办法完成,如果必要的话,修改你的方法以使它充分通用。
36. 找出停止条件,它指示此问题本身或问题的某个适当部分已完成。

*** 分而治之

37. 分而治之是最广泛应用和最强大的算法设计方法之一。当面对程序设计问题时,首先通过求解两个(或更多)的同样形式但规模较小的问题来了解是否能得到问题的解决方案。如果可以的话,就可用分而治之的方法来表达算法并使用递归来设计程序。
38. 非常仔细地保证你的算法总会终止,并正确地处理简单的情况。

*** 分析:递归树

39. 分析递归算法的关键工具是递归树。画出适合于问题的一或两个简单例子的递归树。
40. 应对递归树加以研究,以发现递归是否不必要地重复工作,或者递归树是否表达了任务的有效划分。
41. 递归函数能正确地实现使用栈的迭代函数所完成的同样任务。仔细考虑是递归还是带栈的迭代将产生更清晰的程序并更深入地了解问题。

*** 尾部递归

42. 如果空间因素重要的话,可以消除尾部递归。
43. 递归总是能够转换成迭代,但这条常规经常造成的后果是程序结构十分晦涩难懂。只有当程序设计语言本身无法避免产生这种晦涩时,我们才能容忍它,并且要求它有良好的文档。
44. 研究你的程序,看看它是否适合递归算法的一个标准范例,如分而治之、回溯或树结构的算法。
45. 使递归的使用适合问题的结构。当问题的条件能被完全理解时,所要求的算法的结构将更易于领会。

D.3 数据结构的设计

46. 让数据构建程序,同时提炼算法和数据结构。

47. 一旦数据结构化，算法就几乎形成了。

48. 使用数据结构阐明程序的逻辑。

*** 表和表格

49. 在考虑详细的结构之前，决定数据上要求什么操作，且使用这个信息决定此数据是属于表还是表格。数据结构的遍历或以预先指定的次序访问所有的数据通常隐含着选择表结构，而以时间 O(1)去访问任何元素通常隐含着选择表格。

*** 信息隐藏，封装

*** 自顶向下设计

50. 在实现数据结构时使用信息隐藏和封装：使用函数访问数据结构，并且将它们留在类中并与应用程序分离。

51. 如同设计算法，对你的数据结构使用自顶向下的设计。首先确定数据的逻辑结构，然后慢慢地说明更多的细节，并尽可能延缓实现决策。

52. 尽可能推迟实现数据结构细节的决定。

53. 避免使用错综复杂的方法存储数据，错综复杂通常不能推广到新的情形。

54. 选择实现前，确信已在抽象层充分地描述所有的数据结构及其相关的操作。

55. 使用信息隐藏：将数据结构的应用和它们的实现分离。

56. 选择实现前，确信所有的数据结构和相关联的操作在抽象层充分说明。

*** 实现

57. 在多种实现间进行选择，考虑数据结构上必需的操作。

*** 派生类

58. 如果类 A 的每个对象都有类 B 的对象所有的特征，则将 A 作为 B 的派生类实现。

59. 当声明一个基类的成员时，考虑派生类的要求。

*** is-a 和 has-a 关系

60. 通过使用公共继承实现类间的 is-a 关系。

61. 通过分层实现类间的 has-a 关系。

*** 继承

62. 使用私有继承去模拟类间的“用……实现”的关系。

*** 极端情况

63. 总是验证你的算法对空的结构和仅有一个结点的结构能正确地工作。

D.4　算法设计和分析

*** 前置和后置条件

64. 编写的每个程序、函数和方法要包含准确的前置条件和后置条件。

65. 不要只见树木不见森林。

*** 问题说明

66. 确信你完全地理解了问题。如果必须改变其条件，则确切地解释所做的修改。

*** 用户接口

67. 最精心地设计用户接口，程序的成功很大程度上是靠它的吸引力和易用性。

*** 简化

68. 尽你所能保持算法简单。当犹豫不决时，选择简单的方式。

69. 有时延缓问题会简化解决方案。

70. 保持逻辑简单。

71. 保持函数简短，任何函数很少超过一页长。

72. 设计算法时选择数据结构，避免做出不成熟的决定。

73. 避免为了复杂而复杂，如果简单的方法已足够于你的应用，则使用简单的方法。

74. 避免做重复的工作，如果现成的类模板或函数已足够于你的应用，则使用现成的函数。

*** 算法验证

75. 开始编码前确信算法是正确的。

76. 验证算法中复杂的部分。

77. 万一困难，明确循环的每次迭代前后语句都正确，并验证它们成立。

78. 确信在决定如何求解问题之前理解它。

79. 确信在开始程序设计前理解算法的方法。

80. 万一困难，则将问题划分成多块并分别考虑每一部分。

*** 泊松分布

81. 使用泊松随机变量模拟随机事件的发生。

*** 极端情况

82. 对极端的情况总是保持小心，确信你的算法在到达任务结尾时能够优雅地终止。

83. 在设计算法时，小心极端情况，如空表、仅有一个元素的表或满表(在顺序情况下)。

*** 算法分析

84. 画树是一个优秀的方法，可以同时用于跟踪算法的活动和分析它的行为。

85. 对大型应用依赖于算法的大-O分析，但对小型应用则不要。

*** 哈希表分析

86. 回忆关于哈希的分析，某些冲突的发生是不可避免的，因此当键在表格中接近均匀分布时，不必担心冲突的存在。

87. 对开放定址来说，直到哈希表超过半满，聚集才会成为一个问题。如果表格比记录需要的空间大几倍，则线性探测将足够了；否则需要更复杂的冲突解决方案。另一方面，如果表格比所需的大许多倍，则所有未用空间的初始化可能要求过多的时间。

D.5 程序设计

88. 在说明准确和完善前不要进行编码。

89. 匆匆行动，慢慢后悔。匆匆编程，常常调试。

*** 命名

90. 最审慎地选择类、变量和函数的名称，并予以详尽的解释。

91. 在描述问题时产生的名词为问题的解答提议了有用的类，动词则提议了有用的函数。

*** 文档

92. 在编写每个函数时包含细致的文档(如1.3.2节)。

93. 确信为每个函数写下准确的前置条件和后置条件。
94. 保持文档简练但具有描述作用。
95. 阅读程序的时间比编写程序的时间多得多，使阅读更容易。

*** 类和函数

96. 使用类来模拟程序的基本概念。
97. 每个函数应该仅仅完成一项任务，但要很好地完成。
98. 每个类或函数应该隐藏某些东西。
99. 数据结构的公共方法应当不带前提地予以实现，而数据成员应当保持私有。

*** 全局变量

100. 保持连接简单。只要可能，避免使用全局变量。
101. 只要能够避免，切勿引起副作用。如果必须使用全局变量作为输入，则详细地将它们写入文档。

*** 模块化输入和输出

102. 将输入和输出作为独立的函数，使得它们易于修改并能自定义修改以适应计算系统。

*** 错误检查

103. 在函数开始时包含错误检查以证明前置条件确实成立。
104. 在每次使用函数前，自问为何知道其前置条件会被满足？
105. 确信初始化数据结构。
106. 设计算法时，总是小心极端情形并得体地处理它们。跟踪算法以判断在极端情况下将会发生什么，特别是当数据结构为空或者满的时候。
107. 做尽可能彻底的错误检查，确信函数在其前置条件中规定了所要求的每个条件，并且虽然如此，尽可能方便地避免函数受到许多违反其前置条件的情况的影响。
108. 确信所有变量被正确地初始化。
109. 加倍努力地检查循环的终止条件，并确信循环总会向终止进展。

D.6　用指针对象进行程序设计

*** 选择链式或顺序

110. 在选择链式和顺序实现时，考虑数据结构上的必要操作。链式结构在插入、删除和重排操作中更灵活，而顺序结构有时候更快。
111. 当结构中的项较小且算法简单时，顺序结构一般需要较少的内存、计算机时间和程序设计工作。当结构存放较大的记录时，链式结构通常更节省空间、时间和程序设计工作。
112. 动态内存和指针允许程序自动地适应大范围的应用规模，而且在不同数据结构的空间分配上提供了灵活性。自动内存对完全可以预先描述规模的应用有时更有效。

*** 指针引用

113. 未初始化或随机的指针总是应该重新设置为 NULL。指针对象在删除后应该重新设置为 NULL。
114. 在对指针重新赋值前，确信它所引用的对象不会变成垃圾。

*** 安全

115. 链式数据结构应当实现析构函数、拷贝构造函数和重载赋值运算符。

* * *设计链结构

116. 画出链式结构适当部分的“之前”和“之后”图，显示相关指针及其应被修改的方式。如果有帮助的话，也画图显示此过程的中间阶段。

117. 确定值应以何种顺序放入指针域以实现各种修改，通常最好先将值赋给先前未定义的指针，然后赋给值为 NULL 的指针，最后再赋给剩下的指针。在一个指针变量被复制到另一个指针变量后，可以自由地重新分配第一个指针变量以指向新的位置。

* * *未定义链

118. 在链式结构的方法结束之际，确信没有未定义的链，如新结点中从未赋值的链，或者是旧结点中已变为悬浮的链，即它指向不再使用的结点。应该将这样的链重新分配给仍旧在用的结点或者将它的值设为 NULL。

* * *多重引用解除

119. 避免使用(p->next)->next 这样的构造，即使它们在语法上是正确的。单个对象应仅仅包括单个的指针引用解除。对重复解除的结构，通常预示着可以通过重新思考算法中应声明什么样的指针变量来加以改进，如果有必要就引进新的变量。

D.7 调试和测试

120. 测试数据的质量比数量更重要。

121. 程序测试可用于说明 bug 的存在，而不能说明其不存在。

122. 使用占位程序和驱动器、黑盒和玻璃盒测试来简化调试。

123. 使用大量的支架来帮助定位错误。

124. 在带数组的程序设计中，谨慎比 1 小的下标值，总是使用极值测试来检查使用数组的程序。

125. 在编写程序时保持良好的格式——它会使调试更加简单。

126. 保持文档与代码一致，而且在阅读程序时确信是在调试代码而不仅仅是注释。

127. 向其他人解释程序，这样做会有助于自己更好地理解程序。

128. 客户程序在使用了一个类方法后，应决定是否检查因而发生的错误状态。类应该设计成让客户程序来决定如何响应错误。

D.8 维护

129. 对一个大型且重要的程序，超过一半的工作是在它已被完全调试、测试并投入使用后，来自于维护阶段。

* * *优化

130. 除非必要，不要优化代码。在代码完善和正确之前，不要开始优化代码。

131. 大多数程序将 90%的时间花在 10%的指令上，找出这 10%，集中精力提高它的效率。

132. 为改进程序，复查它的逻辑。不要基于低劣的算法优化代码。

133. 在程序正确且能够运行之前决不要优化它。

134. 不要优化代码，除非绝对必要。

135. 在代码完善地运行前不要优化它，仅在确实需要提高效率的情况下优化代码。首先尝试数据结构的简单的实现，仅在简单的数据结构实现证明效率低下时才换用更复杂的实现。

＊＊＊原型

136. 总是计划建立原型并丢弃它，不管是否计划都必须这样做。

137. 重新开始经常比给一个旧程序打补丁更简单。

术 语 表

abstract data type 抽象数据类型
adjacency list 邻接表
adjacency table 邻接表格
ADT(abstract data type) 抽象数据类型
algorithm 算法
algorithm analysis 算法分析
alpha-beta pruning α-β剪枝
amortized analysis 平摊分析
amortized cost 平摊代价
array 数组
asymptotic 渐近
atomic type 原子类型
automatic object 自动对象
average behavior 平均性能
average-case analysis 平均情况分析
AVL tree AVL树
B*-tree B*-树
B-tree B-树
backtracking 回溯
balance factor 平衡因子
balanced multi-way tree 平衡多路树
base class 基类
big-() 大-()
binary operator 二元运算符
binary search tree 二叉查找树
binary search 二分查找
binary tree 二叉树
block diagonal matrix 块对角矩阵
breadth-first ordering 广度优先次序
breadth-first 广度优先
Catalan number Catalan数
circular queue 循环队列
class 类
clustering 聚集
collision 冲突
combinatorial mathematics 组合数学
comparison count 比较次数
comparison tree 比较树
compiler 编译器
computability 可计算性
connected graph 连通路径
constant function 常量函数
constructor 构造函数
contiguous list 顺序表
copy constructor 拷贝构造函数
cubic function 三次函数
cycle 环
data abstraction 数据抽象
data members 数据成员
data structure 数据结构
decision tree 决策树
depth-first 深度优先
deque 双端队列
derived class 派生类
destructor 析构函数
directed graph(digraph) 有向图
disconnected 不连通
discrete mathematics 离散数学
divide and conquer 分而治之
double-order traversal 双序遍历
doubly linked list 双向链表
dummy node 哑结点
dynamic memory management 动态内存管理

dynamic object 动态对象
edge 边
empty subtree 空子树
encapsulation 封装
exception handling 异常处理
exponential function 指数函数
expression tree 表达式树
extended binary tree 扩展二叉树
Fibonacci number 斐波纳契数
Fibonacci search 斐波纳契查找
Fibonacci tree 斐波纳契树
FIFO(first in,first out) 先进先出
folding 折叠
forest 森林
free tree 自由树
friend function 友元函数
general list 通用表
generics 泛型
Glass-Box Method 玻璃盒测试方法
grandparent node 祖父结点
graph 图
greedy algorithm 贪心算法
harmonic number 调和数
hash function 哈希函数
hashing 哈希,散列
heapsort 堆排序
implementation 实现
infix form 中缀式
infix 中缀
information hiding 信息隐藏
information retrieval 信息检索
inheritance 继承
inline 内联
inorder traversal 中序遍历
insertion sort 插入排序
instability 不稳定性
interpolation search 插值查找
interpolation sort 插值排序
inverted table 反转表格
iteration 重复,迭代
jagged table 锯齿表格
key 键,关键字
left child 左孩子
left subtree 左子树
level-by-level traversal 层次遍历
linear probe 线性探测
linear 线性
linked list 链表
linked queue 链式队列
linked stack 链栈
list 表
load factor 装填因子
logarithmic function 对数函数
lower triangular matrix 下三角矩阵
mergesort 归并排序
method 方法
minimal spanning tree 最小生成树
minimax method 极小极大方法
modular arithmetic 模运算
multiway search tree 多路查找树
multiway tree 多路树
nested function call 嵌套的函数调用
network 网络
NP-complete problem NP-完全问题
object-oriented programming 面向对象的程序设计
open addressing 开放定址
operator 运算符
orchard 果园
order of magnitude 数量级
order 阶
ordered list 有序表
ordered tree 有序树
overflow 溢出,上溢
overloaded function 重载函数
overloading 重载
package 软件包
parallel and distributed algorithm 并行和

分布式算法
parents node 双亲结点
path 路径
pigeonhole principle 鸽巢原理
pointer 指针
Polish notation 波兰表示法
polynomial 多项式
postfix form 后缀式
postorder traversal 后序遍历
prefix form 前缀式
preorder traversal 前序遍历
priority 优先级
private 私有
program listing 程序清单
programming principle 程序设计原理
pseudorandom number 伪随机数
public 公用
the Polish notation 波兰表示法
quadratic function 二次函数
quadratic probing 二次探测
queue 队列
quicksort 快速排序
radix sort 基数排序
random number 随机数
random probe 随机探测
record 记录
rectangular table 矩形表格
recursion 递归
recursive algorithm 递归算法
red-black tree 红-黑树
reference 引用
refinement 细化
rehashing 再哈希
reverse Polish form 逆波兰式
reverse Polish 逆波兰
reversing input 反转输入
right child 右孩子
right subtree 右子树
root 根
rooted tree 有根树
rotation 旋转
scan sort 扫描排序
search tree 查找树
search 搜索,查找
seed 种子
selection sort 选择排序
self-loop 自回路
sentinel 监视哨
sequential search 顺序查找
Shell sort 希尔排序
shortest path 最短路径
sort 排序
sparse table 稀疏表格
splay tree 伸展树
stack 栈
STL(standard template library) 标准模板库
string 字符串
strongly connected 强连通
structure type 结构化类型
stub 占位,占位程序,占位函数
subtree 子树
symmetric order 对称次序
table lookup 表格查找
table 表格
tail recursion 尾部递归,右递归
telescoping sum 套叠和
template 模板
three-way tree 三路树
top-down refinement 自顶向下的细化
traverse 遍历
treesort 树排序
Tri-diagonal matrix 三对角矩阵
truncation 截取
tic-tac-toe 井字游戏
unary operator 一元运算符
undecidability 不可判定性
undirected graph 无向图

value type 值类型
vertex 顶点
virtual function 虚函数
visit 访问
weakly connected 弱连通
weight 权
worst-case analysis 最坏情况分析

[General Information]
书名=国外经典教材　　C++数据结构与程序设计
作者=
页数=581
SS号=0
出版日期=

封面
书名
版权
前言
目录
正文